



BEA WebLogic Java Adapter for Mainframe

User Guide

BEA WebLogic Java Adapter for Mainframe 4.1
Document Edition 4.1
October 2000

Copyright

Copyright © 2000 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, ObjectBroker, TOP END, and Tuxedo are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Connect, BEA Manager, BEA MessageQ, BEA Jolt, M3, eSolutions, eLink, WebLogic, WebLogic Enterprise, WebLogic Commerce Server, and WebLogic Personalization Server are trademarks of BEA Systems, Inc.

All other company names may be trademarks of the respective companies with which they are associated.

BEA WebLogic Java Adapter for Mainframe User Guide

Document Edition	Part Number	Date	Software Version
4.1	N/A	October 2000	BEA WebLogic Java Adapter for Mainframe Version 4.1

Contents

About This Guide

Who Should Read This Guide	xii
System Administrators	xii
Java Application Developers	xii
CICS Application Developers	xiii
IMS Application Developers	xiii
How this Guide Is Organized	xiii
Product Documentation	xv
How to Use The Documentation	xv
Document Conventions	xvi
Contact Us	xviii

1. Understanding the BEA WebLogic Java Adapter for Mainframe Solution

The BEA WebLogic™ Enterprise Application Integration Solution	1-2
Enterprise Application Integration (EAI): A Step Beyond the Middleware Solution	1-2
The BEA EAI Solution on WebLogic Server	1-3
Understanding the WebLogic Sever Integration Environment	1-4
Accelerate Your Development Time	1-4
Build For the Enterprise	1-5
Scale and Manage as Your Needs Grow	1-7
WebLogic Application Servers	1-8
WebLogic Server	1-10
WebLogic Enterprise	1-10
WebLogic Express	1-10
BEA WebLogic Java Adapter for Mainframe Overview	1-11

JAM Architecture	1-11
eGen COBOL Code Generator	1-13
Java Communications Resource Manager Gateway	1-14
System Network Architecture Communications Resource Manager	1-14
2. Understanding How JAM Uses XML	
What is XML?	2-1
Document Type Definition.....	2-2
XML Schema.....	2-3
JAM XML Capabilities	2-3
3. Configuring the BEA WebLogic Java Adapter for Mainframe	
General Steps for Defining the Gateway Configuration.....	3-2
About the JCRMGW Configuration File.....	3-2
JCRMGW Configuration File Sections	3-3
JC_REMOTE_DOMAINS Section.....	3-3
Valid Keywords.....	3-3
Keyword Definitions	3-4
JC_SNACRM Section	3-4
Valid Keywords.....	3-4
Keyword Definitions	3-5
JC_SNASTACKS Section.....	3-6
Valid Keywords.....	3-6
Keyword Definitions	3-6
JC_SNALINKS Section	3-7
Valid Keywords.....	3-7
Keyword Definitions	3-8
JC_LOCAL_SERVICES Section.....	3-10
Valid Keywords.....	3-10
Keyword Definitions	3-11
JC_REMOTE_SERVICES Section.....	3-12
Valid Keywords.....	3-12
Keyword Definitions	3-12
Sample Configuration File.....	3-14

Post Configuration Requirements.....	3-15
--------------------------------------	------

4. Developing Java Applications

Building the Base Java Application	4-1
Dataview Concepts.....	4-2
Obtaining the COBOL Copybook.....	4-5
Creating a New Copybook	4-5
Using an Existing Copybook.....	4-6
Script Comments.....	4-7
Script Comments.....	4-10
Generating the Java Application Source	4-11
Generating a Servlet-Only JAM Application.....	4-11
Creating a Script.....	4-12
Processing a Script	4-16
Generated Files	4-16
Customizing a Servlet-Only JAM Application	4-17
Generating a Client Enterprise Java Bean-Based Application.....	4-26
Creating a Script.....	4-26
Processing the Script.....	4-29
Working with Generated Files	4-29
Customizing an Enterprise Java Bean-Based Application.....	4-34
Compiling and Deploying.....	4-40
Generating a Server Enterprise Java Bean-Based Application	4-41
Creating a Script.....	4-41
Processing the Script.....	4-42
Generated Files	4-43
Customizing a Server Enterprise Java Bean-Based Application	4-47
Compiling and Deploying.....	4-51
Generating a Stand-Alone Client Application	4-52
Processing a Script	4-53
Generated Files	4-54
Customizing a Stand-Alone Java Application	4-55
Using Client Diagnostic Features	4-61
Client Traffic Tracing.....	4-62
Client Loopback	4-63

Client Stub Operation	4-63
5. Deploying Applications	
Deploying Servlets	5-1
Deploying Enterprise Java Beans	5-2
6. Security	
7. Programming Reference	
Field Name Mapping Rules	7-2
Field Type Mappings	7-2
Group Field Accessors	7-3
Elementary Field Accessors	7-4
Array Field Accessors	7-5
Fields with REDEFINES Clauses	7-5
COBOL Data Types	7-6
Other Access Methods for Generated DataView Classes	7-8
Mainframe Access to DataView Classes	7-9
XML Access to DataView Classes	7-11
Hashtable Access to DataView Classes	7-12
Code for Unloading and Loading Hashtables	7-13
Rules for Unloading and Loading Hashtables	7-14
Name Translator Interface Facility	7-14
8. Using BEA WebLogic Process Integrator with JAM	
WebLogic Process Integrator Overview	8-2
WebLogic Process Integrator Architecture	8-2
Integration with JAM	8-4
WebLogic Process Integrator to Mainframe Requests	8-4
Using the eGen COBOL Utility for WebLogic Process Integrator to Mainframe Requests	8-5
Workflow Development for WebLogic Process Integrator to Mainframe Requests	8-5
Mainframe to WebLogic Process Integrator Requests	8-7
Example of JAM Application Integrated with WebLogic Process Integrator ...	8-8
Task 1: Set Up JAM Components as Business Operations	8-8

Task 2: Set Up a Workflow	8-13
Task 3: Define the Start Node	8-16
Task 4: Set Up the Start Node Properties	8-20
Task 5: Create a Setup Name Task	8-22
Task 6: Create Additional Tasks	8-28

9. Developing a Multi-Service Data Entry Servlet

Task 1: Use eGen COBOL to Create a Base Application	9-2
Step 1: Prepare eGen COBOL Script	9-2
Step 2: Add Service Entries	9-3
Step 3: Add Page Declaration in eGen COBOL Script	9-3
Step 4: Add Servlet Name	9-4
Step 5: Generate the Java Source Code	9-4
Step 6: Review the Java Source Code	9-5
Task 2: Create Your Custom Application from the Base Application	9-6
Step 1: Start with Imports	9-6
Step 2: Declare the New Custom Class	9-6
Step 3: Add Implementation for doGetSetup	9-7
Step 4: Continue Implementation for doGetSetup	9-7
Step 5: Finish Implementation for doGetSetup	9-8
Step 6: Create Implementation for doPostSetup	9-8
Step 7: Continue Implementation for doPostSetup	9-9
Step 8: Continue Implementation of doPostSetup	9-9
Step 9: Continue Implementation for doPostSetup	9-10
Step 10: Finish Implementation of doPostSetup	9-10
Step 11: Create Implementation for doPostFinal	9-11
Step 12: Update the jcrmgw.cfg File with Service Entries	9-11
Step 13: Create Basic Three-Part HTML Frame	9-12
Step 14: Create a Series of Links to HELP Pages	9-12
Task 3: Update the JAM Configurations and Update BEA WebLogic Server Properties	9-13
Task 4: Deploy Your Application	9-14
Task 5: Use the Application	9-15
Sample COBOL Programs for the Form Buttons	9-17
Create	9-17

Read.....	9-18
Update.....	9-19
Delete.....	9-20

10. Enhancing an Existing Servlet to Originate a Mainframe Request

Task 1: Use eGen COBOL to Create a Base Class	10-2
Step 1: Prepare eGen COBOL Script	10-2
Step 2: Generate the Java Source Code.....	10-3
Step 3: Review the Java Source Code	10-4
Task 2: Update the Survey Servlet Using the Generated Class.....	10-4
Step 1: Start with Imports	10-5
Step 2: Add New Data Members	10-5
Step 3: Update doPost with Mainframe Request.....	10-5
Step 4: Continue Updating doPost by Extracting Form Data.....	10-6
Step 5: Continue Updating doPost by Calling Mainframe Service	10-7
Task 3: Update the JAM Configurations and Update WebLogic Server Properties	10-8
Task 4: Deploy Your Application	10-8
Task 5: Use the Application	10-9
Sample COBOL Program to Write to Temporary Storage Queue	10-10

11. Updating an Existing EJB to Service a Mainframe Request

Task 1: Use eGen COBOL to Create a Base Class	11-2
Step 1: Prepare eGen COBOL Script	11-2
Step 2: Generate the Java Source Code.....	11-3
Step 3: Review the Java Source Code	11-3
Task 2: Update the Trader Interface Using the Generated Class.....	11-4
Step 1: Start with Import	11-5
Step 2: Continue with Imports.....	11-5
Step 3: Update EJB with dispatch	11-6
Step 4: Continue Updating EJB with dispatch	11-6
Step 5: Finish Updating EJB with dispatch.....	11-7
Task 3: Update the JAM Configurations	11-7
Task 4: Deploy Your Application	11-8

Task 5: Use the Application	11-8
Sample COBOL Program to Write to Temporary Storage Queue	11-9

12. Integrating JAM with Crossplex

Task 1: Create a CrossPlex Script	12-2
Step 1: Prepare Inbound Record Definition	12-3
Step 2: Create a Copybook of the Inbound Record Definition	12-5
Step 3: Create an Outbound Record Definition and Copybook	12-5
Step 4: Prepare the CrossPlex Script.....	12-7
Step 5: Test and Debug the Script.....	12-8
Handling the Mainframe Sign-on	12-8
Task 2: Use eGen COBOL to Create a Base Application.....	12-9
Step 1: Prepare eGen COBOL Script.....	12-12
Step 2: Add Service Entry.....	12-12
Step 3: Add Page Declarations in eGen COBOL Script	12-12
Step 4: Add Servlet Name.....	12-13
Step 5: Generate the Java Source Code.....	12-13
Task 3: Create Your Custom Application from the Base Application.....	12-14
Step 1: Start with Imports.....	12-14
Step 2: Declare the New Custom Class.....	12-15
Step 3: Add Implementation for doGetSetup.....	12-15
Step 4: Create Implementation for doPostSetup	12-15
Step 5: Create Implementation for doPostFinal	12-17
Task 4: Update the JAM Configurations and Update WebLogic Server Properties.....	12-18
Task 5: Deploy Your Application	12-19
Task 6: Use the Application	12-19

A. Code Generator Reference Pages

eGen COBOL	A-1
Synopsis	A-1
Script Syntax Reserved Words.....	A-2
General Rules.....	A-3
Grammar.....	A-4
Results of Running the Code Generator.....	A-6

B. Configuration Checker Utility

bea.sna.jcrmgw.jcrmConfigurator	B-2
Synopsis.....	B-2
Description	B-2
GWBOOT	B-3
Synopsis.....	B-3
Description	B-3

C. Error and Informational Messages

D. Java Docs

Glossary

Index

About This Guide

This guide provides information about the BEA WebLogic Java Adapter for Mainframe (JAM), a product that enables client/server transactions between Java applications and OS/390 CICS or IMS programs.

This guide provides you with scenarios for developing and deploying applications by using the software components within the JAM environment. The scenarios depict specific tasks that work in conjunction with development and deployment procedures also provided in this guide. It is best to read and become familiar with the contents of “Developing Java Applications” and “Deploying Applications” before attempting the exercises in “Programming Scenarios.”

To learn more about how the System Network Architecture Communications Resource Manager (SNACRM) provides the emulation that enables CICS DPL protocols to flow into and out of the JAM environment, refer to the *BEA SNACRM Administration Guide*. This guide is accessible from the JAM online documentation in both HTML and PDF formats.

This section covers the following topics:

- [Who Should Read This Guide](#)
- [How this Guide Is Organized](#)
- [Product Documentation](#)
- [Contact Us](#)

Who Should Read This Guide

The audience is primarily Java application developers, Customer Information Control System (CICS) application developers, Information Management System (IMS) application developers, and system administrators who configure gateway software on BEA Web Logic Server (WLS) platforms.

System Administrators

As the application administrator, you install the WLS platform software and configure the JAM gateway. You must have sufficient System Network Architecture (SNA) knowledge to configure the underlying SNA stack (PU2.1 server) so it conforms with definitions created in Virtual Telecommunications Access Method (VTAM) and CICS for each remote domain. Refer to stack vendor documentation for details.

Successfully linking and establishing conversations between Java applications and mainframe-based programs requires special coordination. The names and characteristics of mainframe resources, configured in the SNA stack, must agree with resources and characteristics defined in VTAM and CICS.

Typically, remote VTAM and CICS resources are defined by system personnel in a data center where IBM mainframes are located. Therefore, you need to request the remote names of IMS and CICS resources from data center systems personnel and use those names to configure the SNA stack and the JAM gateway.

Java Application Developers

The JAM facility provides application developers bidirectional, transparent processing using platform native Application Programming Interfaces (API). Java application programmers can develop clients or servers using standard object-oriented programming techniques without regard for mainframe protocols.

CICS Application Developers

CICS programmers can develop servers or clients using the CICS Link command and the Distributed Program Link (DPL) subset API as a standard distributed CICS application. In addition, any existing CICS application designed to be invoked from a DPL can be used without modification.

IMS Application Developers

IMS programmers can use any IMS program (without modification) that sends and receives messages to and from the IMS message queue as either a client or a server.

How this Guide Is Organized

The *BEA WebLogic Java Adapter for Mainframe User Guide* is organized as follows:

- [Understanding the BEA WebLogic Java Adapter for Mainframe Solution](#)
This section gives an overview of the JAM environment and describes two configurations for the product.
- [Understanding How JAM Uses XML](#)
This section explains how JAM uses the capabilities of XML to exchange data between different applications and operating systems.
- [Configuring the BEA WebLogic Java Adapter for Mainframe](#)
This section provides information and procedures for configuring the JAM development environment to create a basic servlet that can execute a CICS DPL.
- [Developing Java Applications](#)
This section explains how to build a base Java application from a COBOL copybook and generate Java application source code.

- [Deploying Applications](#)

This section gives procedures for installing developed servlets and/or EJB into an operational environment.

- [Security](#)

This section describes the basic level of security supported by the JAM product software.

- [Programming Reference](#)

This section provides data mapping rules for the `EgenCobol` tool used to generate Java code from COBOL copybooks.

- [Using BEA WebLogic Process Integrator with JAM](#)

This section gives an overview of how JAM integrates with WebLogic Process Integrator to provide a user modeling tool and workflow environment that combines the functions of static data and server administration, workflow definition, and workflow monitoring.

- [Developing a Multi-Service Data Entry Servlet](#)

This section contains a scenario that shows how to develop a multi-service application, as opposed to the single-service application presented in [Generating a Servlet-Only JAM Application](#).

- [Enhancing an Existing Servlet to Originate a Mainframe Request](#)

This scenario illustrates how to enhance an existing servlet to originate a mainframe request. Use the `WebLogic Serversurvey` servlet and add a mainframe request to the `post` routine.

- [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.

- [Integrating JAM with Crossplex](#)

This section contains a scenario that shows how to develop a single service servlet-based application that invokes a CrossPlex script on the mainframe.

In addition, the appendixes contain the following information:

- [Code Generator Reference Pages](#)

This appendix contains reference pages for the EgenCobol tool.

- [Configuration Checker Utility](#)

This appendix describes the configuration checker utility that can be used to verify the contents of the `jcrmgw.cfg` file before starting the Java Communications Resource Manager Gateway (JCRMGW).

- [Error and Informational Messages](#)

This appendix gives descriptions of messages that may be encountered while using JAM software.

- [Java Docs](#)

This appendix tells how to use the HTML pages describing JAM Java classes.

- [Glossary](#)

The glossary contains definitions of some important terms used in this guide.

Product Documentation

The JAM documentation consists of the following:

- *BEA WebLogic Java Adapter for Mainframe Release Notes*
- *BEA WebLogic Java Adapter for Mainframe Installation Guide*
- *BEA WebLogic Java Adapter for Mainframe User Guide*
- *BEA WebLogic Java Adapter for Mainframe, SNACRM Administration Guide*

How to Use The Documentation

The Documentation CD-ROM included in the package with your product software CD-ROM contains an HTML Web User Interface (WUI). The WUI links to HTML versions and PDF versions of JAM documentation. The WUI should be viewed on an

online browser. The PDF versions should be used for printing. (Information on how to view the online documentation is available in the *BEA WebLogic Java Adapter for Mainframe Release Notes*.)

Note: The WUI requires a Web browser that supports HTML 3.0. Netscape Navigator 2.02 or Microsoft Internet Explorer 3.0 or later.

You must have the Adobe Acrobat Reader to print the PDF file. If you do not have this reader, you can obtain it free of charge from the Adobe Systems Incorporated home site at www.adobe.com. (The BEA WebLogic Java Adapter for Mainframe WUI contains a hot link to this site.)

Document Conventions

The following documentation conventions are used throughout this manual:

Convention	Item
blue text	Indicates hypertext link to related topic in PDF file or HTML document.
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
<i>italics</i>	Indicates emphasis or book titles.
monospace text	<p>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.</p> <p><i>Examples:</i></p> <pre>#include <iostream.h> void main () the pointer psz chmod u+w * \tux\data\ap .doc tux.doc BITMAP float</pre>

Convention	Item
monospace boldface text	Identifies significant words in code. <i>Example:</i> void commit ()
<i>monospace</i> <i>italic</i> <i>text</i>	Identifies variables in code. <i>Example:</i> String <i>expr</i>
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. <i>Examples:</i> LPT1 SIGNON OR
{ }	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
[]	Indicates optional items in a syntax line. The brackets themselves should never be typed. <i>Example:</i> buildobjclient [-v] [-o name] [-f <i>file-list</i>]... [-l <i>file-list</i>]...
	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
...	Indicates one of the following in a command line: <ul style="list-style-type: none"> ■ 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. <i>Example:</i> buildobjclient [-v] [-o name] [-f <i>file-list</i>]... [-l <i>file-list</i>]...
.	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contact Us

If you have any questions about this version of BEA WebLogic Java Adapter for Mainframe, or if you have problems installing and running the software, 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, which 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 products you are using
- A description of the problem and the content of pertinent error messages

1 Understanding the BEA WebLogic Java Adapter for Mainframe Solution

Businesses around the world are using BEA WebLogic Server as the deployment platform for Java-based e-commerce applications. These applications support a wide range of business transactions, from simply displaying the content of a web page to processing secure, online financial transactions. As the complexity of web-based applications evolve, the business transactions supported by these applications may require direct integration with IBM mainframe applications in order to complete. Storing up transactions for the mainframe and executing them periodically in batch mode is not sufficient. To meet the rigorous demands for fast and flawless execution of customer service requests, the business transactions must integrate with mainframe applications in real-time. BEA WebLogic Java Adapter for Mainframe (JAM) built on WebLogic Server meets these demands.

This section discusses the following topics:

- [The BEA WebLogic™ Enterprise Application Integration Solution](#)
- [BEA WebLogic Java Adapter for Mainframe Overview](#)

The BEA WebLogic™ Enterprise Application Integration Solution

Corporations have become increasingly dependant on technology, investing heavily in top applications and business models to meet their specific business needs. As more complex and disparate technology is added to the business environment, the need for a method of integrating these applications into a unified set of business processes has become increasingly important. Companies have invested millions of dollars in “best-of-breed” business models that are unable to work together as a unified system: The company’s inventory control business application, human resources application and sales automation applications operate in unique ways and cannot communicate with each other without complex and often tenuous connections.

Instead of developing new applications that may or may not function as well as top-of-the-line prepackaged business applications, business managers are demanding that seamless bridges be built between existing business applications to bind them into a single, unified enterprise application. **Enterprise application integration (EAI)** provides the solution to this problem by allowing the unrestricted sharing of data and business processes among any connection applications and data sources in the enterprise without the necessity of changing the original applications or data structures.

Enterprise Application Integration (EAI): A Step Beyond the Middleware Solution

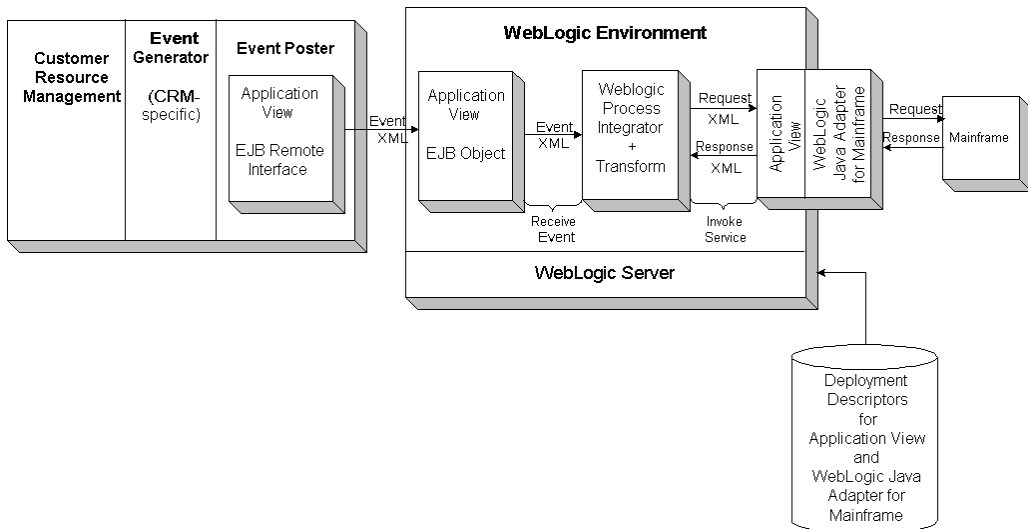
The need for integration between various enterprise applications has historically been solved with middleware, which provides only point-to-point linkage that requires complex links and significant alterations to source code and target systems. An EAI solution is a platform that allows all transaction information to be immediately available in any form and from any location in the enterprise and makes every application, database, transaction table, method available at any time. It is implemented by providing a set of integration-level application semantics that provide a common way for business processes and data to communicate across applications.

This solution focuses on the integration, reuse, and distribution of both business-level processes and data, and it seeks to mask the underlying configuration and functional complexities from users so that they need only an understanding of the applications they are integrating, not the integration system itself. A new generation of application servers allow developers to build and reuse business processes

The BEA EAI Solution on WebLogic Server

Integration requires an open, business-level view of the data and processes in an Enterprise Resource Planning (ERP) system in which the integration points are based on business objects, such as invoice, employee, customer, sales order, and purchase order. Leveraging various interfaces in an integration solution requires complex middleware technology and target application expertise. [Figure 1-1](#) provides a graphic overview of the components of the BEA EAI Solution on WebLogic Server.

Figure 1-1 The BEA EAI Solution on WebLogic Server



Currently, most advanced middleware vendors provide adapters to one or more of these interfaces. However, these offerings typically solve only a particular type of integration problem and usually introduce more proprietary technology into the integration solution. Consequently, the ideal integration solution is an industry

standards-based abstraction layer that hides the complexities of the various interfaces and provides an open, business-level view of your third party applications. The BEA EAI leverages the strength, flexibility, and scalability of BEA WebLogic Server™. [Figure 1-1](#) provides a visual over view of the BEA EAI Solution on WebLogic Server.

Understanding the WebLogic Sever Integration Environment

BEA WebLogic Sever™ is an open, extensible, standards-based application server platform that is used for assembling, deploying, and managing distributed applications. The BEA WebLogic™ product group includes application servers and related products that are designed to solve the urgent IT challenges of e-commerce and Web-ready applications. BEA WebLogic products provide the maximum flexibility in component-based application development, deployment, and management, while protecting IT investments through their unmatched adherence to industry standards. WebLogic is a rich, high-performance, fully integrated Java platform for enterprise applications. With WebLogic, you can:

- [Accelerate Your Development Time](#)
- [Build For the Enterprise](#)
- [Scale and Manage as Your Needs Grow](#)

Note: Refer to the BEA WebLogic™ product documentation on the Product Documentation page at <http://e-docs.bea.com> for the most up-to-date information on WebLogic solutions.

Accelerate Your Development Time

WebLogic isolates your developers and applications from disparities among Java platforms and database interfaces, and from low-level programming complexities like sockets and thread management. These kinds of resources belong in the server infrastructure, not in your application. WebLogic helps you get your application in users' hands quickly in the following ways:

- **Builds reusable components easily** by supporting both Enterprise JavaBeans and classic JavaBeans.

- **Develops server-side Java for your website** by supporting Java-standard Servlets for dispatching Java business logic in HTTP and dynamically constructing HTML responses.
- **Supports more users with scalable RMI** by offering the first high-performance implementation of Java Remote Method Invocation (RMI), over a multiplexed, bi-directional connection for optimal performance and scalability. With RMI, an application can invoke methods on objects stored in any remote JVM as if those objects existed locally.
- **Reduces client overhead with multitier JDBC** by allowing Java applications, including Internet applets, to use a single JDBC-compliant interface to access multiple databases. With pure Java clients, native code is not needed on the client machine. WebLogic JDBC enhances performance with data caching and multiplexed database connections, and it supports DBMS security and transactions.
- **Simplifies event notification and management** by offering a true “server-push” event model—as efficient, subject-based event notification solution for applications needing near-realtime information about changing conditions. An event-enabled application no longer needs to continuously check a connection in order to receive notification of remote events. When an event is submitted, WebLogic either notifies clients that have registered interest in the event or performs an action on behalf of those clients. Server-side content-based filtering allows you to limit interruptions on the client.

Build For the Enterprise

You want to build applications that last. You want to build them fast, and you want to leverage your existing databases, applications, systems, and infrastructure. To accomplish these goals, BEA WebLogic helps you:

- **Develop business components with Enterprise JavaBeans** to provide a component model that uses entity and session beans, which means WebLogic automates database components (entities), as well as classic transaction model components (sessions). WebLogic EJB offers you:
 - Automatic persistence (database and file access)
 - Automatic, declarative transaction models
 - Client authentication and access control at the method level
 - Resource management for threads, network, and database connections

- Bean caching
 - Bean life-cycle management for creating, finding, and destroying beans
 - Concurrency control
 - External configuration of bean runtime properties
- **Support all clients with the same business components.** WebLogic supports multiple programming models: web and intranet-based HTML clients, browser-based applets, intranet and LAN-based desktop clients, as well as legacy environments, such as COM/ActiveX and CORBA.

Standard Java Servlets can isolate the presentation logic from the EJB business logic for the HTML client so that desktop applications written in Java, Visual Basic, PowerBuilder, and other IDEs can access EJB components directly, providing identical business logic to these clients. Combining EJB with the multiple client models supported by WebLogic gives you complete flexibility in choosing and mixing application models.

- **Increase reliability with transparent fault tolerance using WebLogic clusters.** In addition to transparent scaling with load balancing across multiple servers for JNDI-based services, clusters provide redundancy that allows application clients to continue functioning even in the event of a server or network failure. In the WebLogic model, non-persistent conversational and stateless JNDI-based services are essentially immune to server or JNDI instance failures.

WebLogic offers a variety of load-balancing and failover models that can be configured to meet the needs of the application. WebLogic's intelligent "SmartStubs" for RMI and EJB objects can select any one of the servers in a cluster during a name binding. Servlet session management is handled through the standard Java Session Management API rather than a proprietary solution.

- **Integrate with existing applications and databases** by facilitating EJB integration of legacy applications and databases into new applications. EJBs can "wrap" non-Java applications to make it easy to migrate heterogeneous applications to an easy-to-maintain standard. Once a legacy application is wrapped inside Enterprise JavaBeans, the business logic of the legacy application can be converted to Java when business needs dictate, without impacting new applications or clients.

Similarly, Enterprise JavaBeans can access existing databases directly or through legacy transaction systems to provide data and processing to new applications in a component-centric paradigm. As you migrate your legacy applications to Java

or to new systems, the Enterprise JavaBeans insulate client applications from changes.

- **Automate database and transaction programming** by providing automatic database access for bean developers. Using EJB "container-managed" persistence, the WebLogic EJB container provides the mapping from the user-developed bean to the underlying database. Datatype conversion, if any, is automatic. WebLogic also automatically handles transaction boundaries. The EJB developer specifies how the bean should relate to a transaction—start a transaction, participate in a transaction, and ignore a transaction.
- **Lower costs with reusable components** by offering multiple programming and component models. By designing applications with distinct presentation, business logic, and data/legacy application integration tiers, you can achieve a high degree of leverage and reuse. For example, common session Enterprise JavaBeans that contain the logic for business transactions can be shared by HTML clients, Java application clients, Visual Basic/C++ clients, and others.

The same Enterprise JavaBeans can be used in many variants of the application. And because business process information has been abstracted away from the specifics of a single application interface, those business components can be reused over and over again in new applications, to significantly reduce cost over the life cycle of your enterprise applications.

Scale and Manage as Your Needs Grow

Scale and manage WebLogic as your needs grow with the following features:

- **Zero administration clients (ZAC)** makes it very easy to deploy client-side code to users. When you use a WebLogic server as an application dispenser, authorized users can use any browser to request the desired application from WebLogic. When the client-side application is installed, it is automatically updated with any changes you publish through the WebLogic Publisher Wizard, a graphical, easy-to-use interface that takes you step-by-step through the publishing process. Your end-users do not have to manually install anything.
- **Graphical management console** for monitoring and managing a WebLogic configuration. You can see what your WebLogic Servers are doing, where the hot spots are, what your dynamic runtime environment looks like, and what resources specific applications or users are consuming.

- **Security** that provides optional support for Secure Sockets Layer (SSL) to ensure the integrity and privacy of network communications. Security services support X.509 certificates and access control lists (ACLs) to authenticate participants and manage access to network services. WebLogic includes the RSA security algorithms to ensure SSL compatibility with industry-standard browsers.
- **Standard Internet protocols** that allow network communications to flow over HTTP or CORBA IIOP, which can be used across firewalls or to connect CORBA-enabled clients or servers.
- **Global naming.** WebLogic uses the Java-standard JNDI interface with whatever LDAP-compliant directory technology your corporation chooses for managed objects, users, database connections and other resources under its control.
- **Load balancing** that permits server-side components to be dynamically relocated across machines for load leveling. Requests can be balanced across multiple servers that are providing EJB or RMI components, or running servlets for web clients.
- **Fault tolerance.** In a WebLogic cluster, requests for JNDI, EJB, or RMI automatically failover to another cluster member in the event of an individual server failure. Since HTTP sessions can be saved to a persistent store as a database or file system, WebLogic can provide long-lived sessions as well as fault-tolerant sessions to a web community of users.
- **Exception logging** that automatically logs diagnostic information and provides interfaces for applications to log their own exception conditions. Logs can be viewed remotely from a web browser using common log format.

WebLogic Application Servers

An *application server* is the engine that runs the applications that drive e-commerce. Application servers are software programs deployed on high-performance computers that act as go-betweens and service enablers, connecting browsers on desktops with the back-end systems and databases of e-businesses over the Internet. As companies increasingly leverage intranets and extranets as well as the Internet as new business environments, their networked applications need to be dynamically linked to other applications and data sources, and easily deployed and managed across machines and networks that are themselves in flux.

With the industry's most comprehensive suite of transaction and application servers, BEA provides an end-to-end transaction platform for building a solid, easy-to-manage e-commerce system engineered for rapidly launching high-value services to increase competitive advantage.

BEA's transaction and application server family provides development and deployment solutions for e-commerce that stretch from front-end, simple HTML-based Web sites to mainframe-class systems. The marriage of BEA's application server and transaction processing server is a perfect combination of capabilities for the Internet Economy. BEA WebLogic Server™ focuses on Web-based Java applications, while BEA eLink Platform and BEA WebLogic Enterprise™ focus on managing high-volume, heterogeneous transactions. Together, they drive end-to-end e-business with unfailing reliability that both consumers and companies can count on.

The BEA WebLogic family of application servers includes:

- WebLogic Server
- WebLogic Enterprise
- WebLogic Express

Figure 1 WebLogic Family of Application Servers



WebLogic Server

The award-winning BEA WebLogic Server™ provides the performance, scalability, and high-availability required to power e-commerce and Internet business solutions. The server of choice for many e-businesses, it offers the following features:

- **Standards leadership.** WebLogic Server provides an industrial-strength set of services for building e-commerce applications using the Java 2 Enterprise edition (J2EE) standards, including enterprise JavaBeans (EJB) components. BEA WebLogic Server offers the most comprehensive implementation of Enterprise Java standards in the industry.
- **Unlimited Scalability.** WebLogic Server provides a highly scalable architecture featuring client connection sharing, resource pooling, and sophisticated clustering of both dynamic web pages and EJB components.
- **Rapid Development.** WebLogic Server is designed for enterprise e-business applications that demand the flexibility and security of server-side components in Java, while supporting the scalability, performance, and reliability essential to mission-critical solutions. WebLogic Server simplifies development of portable and scalable applications and provides rich interoperability with other applications and systems.

WebLogic Enterprise

For e-commerce solutions that require the greatest choice of development and deployment options and unlimited scalability, BEA WebLogic Enterprise™ delivers a proven and scalable transaction platform that brings together the top industry standards with the power of component-based programming to create complete production-ready e-commerce solutions.

WebLogic Express

BEA WebLogic Express™ provides a solution for developing simple HTML front-end applications to a database for companies who want to develop new business logic in Java but not EJBs or components.

BEA WebLogic Java Adapter for Mainframe Overview

BEA WebLogic Java Adapter for Mainframe (JAM) is a set of software components that provides seamless bidirectional interactions between Java applications running on a WebLogic Server platform and either Customer Information Control System (CICS) applications, or Information Management System (IMS) applications running on a mainframe. With CICS, the Java application request/response operations interact using Distributed Program Links (DPL). With IMS, the Java application request/response operations interact using IMS implicit Application Program-to-Program Communication (APPC) support.

JAM Architecture

BEA WebLogic Java Adapter for Mainframe enables bidirectional connectivity between Java applications running on the WebLogic Application Server and Mainframe COBOL applications running in the CICS or IMS environments.

The JAM product provides the following features:

- Development tools

The eGen COBOL Copybook utility translates COBOL copybooks to JAVA source that can be used as the basis for application development and data transformation.

- Runtime environment

The Java Communications Resource Manager Gateway (JCRMGW) runs in a WebLogic Server instance and an SNA Communications Resource Manager (SNACRM) can run in the native UNIX/NT environment or distributed to the mainframe as an MVS APPC application. The SNACRM may also be distributed to another UNIX/NT system, separate from the WebLogic Server platform.

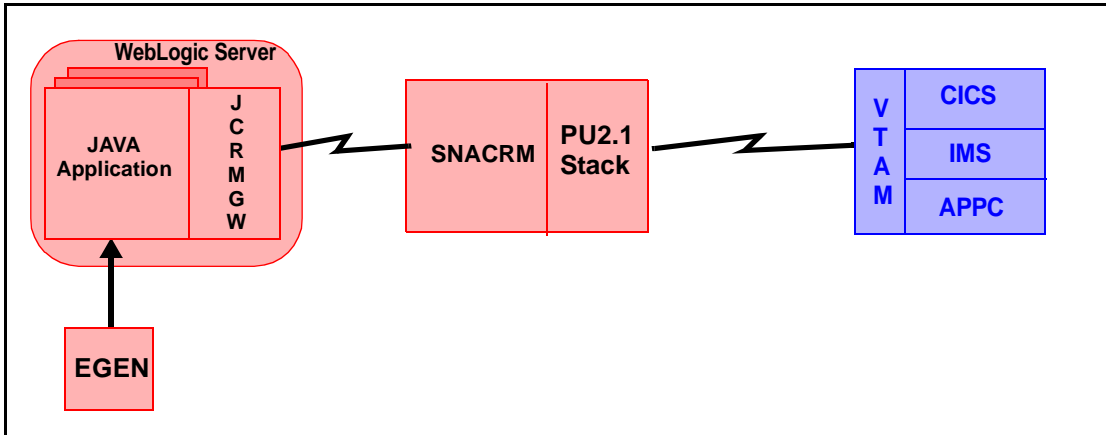
Note: WebLogic Server is required and a third-party SNA stack is required if the SNACRM is not installed on a mainframe, but are sold separately.

In a typical scenario, existing mainframe COBOL applications are exposed to new or existing applications in the WebLogic Server environment. The development tools generate JAVA code from COBOL copybook source. The COBOL copybook contains the structure and size of the data that is passed between the WebLogic Server and mainframe applications. The source code generated from the copybook provides classes that can be extended by the application developer to perform discrete business operations. During runtime, the data passed between applications is stored in a data view object. This object provides set and access methods for all fields in the data buffer. The object also provides a method for converting the object to an XML document. The XML document can be used as input to other applications and, specifically, the WebLogic Process Integrator work flow engine. An XML schema can be generated from the original COBOL copybook for manipulation and transformation of XML documents.

The JAM architecture requires APPC-based communications with mainframe CICS and IMS. The product structure is flexible enough to allow TCP access to the mainframe, if certain product components are installed on the mainframe.

The JCRMGW runs in WebLogic Server and is an eGen client gateway that communicates over TCP/IP to the SNACRM. The SNACRM can run on the same or separate platform other than the JCRMGW. When the SNACRM is running in the UNIX/NT environment, a third party SNA stack is required to enable SNA communications with the mainframe. The SNACRM could alternatively be installed on the mainframe running as a UNIX application under OS/390 UNIX, or as a native MVS APPC application. When the SNACRM is running under OS/390 UNIX or MVS, it does not require a third party SNA stack, but directly accesses VTAM for communications with CICS or IMS. [Figure 1-2](#) illustrates the JAM architecture.

Figure 1-2 JAM Architecture



eGen COBOL Code Generator

The eGen COBOL code generator is the utility that generates Java source code from a COBOL copybook. A base Java application is provided on the JAM product CD-ROM so you can modify it to create custom Java applications. The generated code is dependent on the generation model chosen:

- Servlet only
- Client EJB (with optional servlet)
- Server EJB
- Java Class

The generated code provides for data accessors and conversions as well as miscellaneous functions. The generated servlet provides a basic form that matches the copybook. You can use this servlet for testing or proof of concept. For examples of using the eGen COBOL code generator, refer to [“Developing Java Applications”](#) and [“Programming Scenarios.”](#)

Java Communications Resource Manager Gateway

The Java Communications Resource Manager Gateway (JCRMGW) is the Java application that manages sessions providing access into and out of the Java environment. JCRMGW is an EJB component that runs on WebLogic Server Java Virtual Machine (JVM). JCRMGW processes Java-to-mainframe requests and responses in conjunction with the SNACRM. Requests coming from the mainframe are mapped to an EJB that services the request while requests going to the mainframe are mapped to a mainframe program that can be executed using a CICS DPL or started from an IMS queue.

JAM includes a utility that allows system managers to monitor the status of connections to the mainframe and to set traces on connections for diagnostic purposes.

System Network Architecture Communications Resource Manager

The SNACRM runs as a separate native process. It enables APPC conversations and DPL protocols to flow into and out of the Java environment and runs on the same platform as the SNA stack. The SNACRM obtains its configuration from the JCRMGW. If the Java gateway is running on a platform other than the one on which the SNACRM is running, the SNACRM should be started before WebLogic Server is started so it is monitoring the address specified in the JCRMGW configuration.

The SNACRM supports non-transactional IMS programs using the implicit APPC support for IMS. Implicit APPC is similar to the CICS/ESA DPL. Any IMS program that sends and receives messages to and from the IMS message queue can be used without change as either a client or a server.

In order for the SNACRM to properly operate within the JAM environment, the CICS remote domain must be prepared to conduct operations with the local domain. This includes:

- Establishing the Virtual Telecommunications Access Method (VTAM) configuration
- Configuring the CICS Local Unit (LU)
- Completing cross-platform definitions
- Setting stack traces

For more detailed information, refer to the *BEA SNACRM Administration Guide* accessible from the JAM online documentation in both HTML and PDF formats.

Supported Third-Party SNA Stack

A properly configured SNA protocol stack is required for the SNACRM to communicate with a mainframe if your SNACRM is not installed on the mainframe. (Refer to the *BEA WebLogic Java Adapter for Mainframe Release Notes* for a complete list of supported stacks.) The JCRMGW requires the following parameters from the SNA stack configuration:

- Local LU

The local name of the SNACRM that is used by the mainframe CICS region.

- Remote LU

An alias representing the mainframe partner application to be used by the SNACRM.

- Mode name

SNA mode definitions provide session characteristics for given local/remote LU pair.

- MAX Sessions (optional)

Sessions are required for connections to be made between two LU. MAX sessions should be high enough to handle the expected gateway traffic.

- Minimum Contention Winners (optional)

This parameter is optional and determines priority of session activation.

2 Understanding How JAM Uses XML

BEA WebLogic Java Adapter for Mainframe uses the capabilities of XML to exchange data between different applications and operating systems. Understanding basic XML terms will help you to understand JAM's XML capabilities and how they are used.

This section discusses the following topics:

- [What is XML?](#)
 - [Document Type Definition](#)
 - [XML Schema](#)
- [JAM XML Capabilities](#)

What is XML?

Extended Markup Language, or XML, is a text format for exchanging data between different systems. It allows data to be described in a simple, standard, text-only format. Since the data is presented in a standard form, applications on disparate systems can interpret the data using simple text parsing tools, instead of having to interpret data in proprietary binary formats.

XML documents come in two varieties: data and metadata.

- XML Data Document

Data records can be converted into XML documents, which can then be transmitted to other applications. The XML data documents contain a single top-level entity (or tag) that represents the entire data record. Fields within the record are represented by other subordinate entities nested within the top-level entity. Each entity has a unique tag name, which corresponds to a field within the original data record. Each entity has content, which is the actual data value of the field. Entities may also have attributes, which are values attached to the entities that augment the normal content values. The XML data document file name ends with a .xml extension.

See [Listing 2-2](#) for an example XML data document.

■ XML Metadata

Every XML document consists of a top-level entity, which in turn may be composed of subordinate entities. The structure of these entities, which included their tag names, the order in which they occur, the type and length of their content values, and their allowed attribute values, is described by a metadata definition. Metadata definitions can take the form of XML documents themselves. There are two standard formats for XML metadata documents: XML Document Type Definition (DTD) and XML Schema.

Document Type Definition

A document type definition, or DTD, defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements (tags). While XML provides an application independent way of sharing data, the DTD provides a common definition for interchanging data.

Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

The XML DTD file name ends with a .dtd extension.

See [Listing 2-4](#) for an example XML DTD document.

XML Schema

A schema specifies the structure of an XML document and constraints on its content. While XML is the meta-language that provides the rules for defining tag languages, an XML Schema document is a formal specification of the grammar for a particular tag language. The schema defines the elements that can appear within the document and the attributes that can be associated with an element. It also defines the structure of the document: which elements are child elements of others, the sequence in which the child elements can appear, and the number of child elements. It defines whether an element is empty or can include text. The schema can also define default values for attributes.

XML Schema is more precise than DTD, providing more descriptive information about each XML element. It is likely that XML Schema will eventually replace XML DTD as the dominant standard metadata format.

A schema is useful for validating the document content to determine whether a document is a valid instance of the grammar expressed by that schema and for describing your grammar for use by others.

The XML Schema file name ends with a .xsd extension.

See [Listing 2-3](#) for an example XML Schema document.

JAM XML Capabilities

The JAM eGen tools provide the ability to generate both XML Schema and XML DTD documents for a given COBOL copybook record definition.

The JAM runtime environment provides the capability of converting data records into XML data documents, formatting according to their corresponding schema or DTD definitions.

The following listings show examples of the XML files generated by the eGen COBOL Copybook Utility from the COBOL Copybook for an employee information record.

[Listing 2-1](#) shows an example of an employee information record from a COBOL Copybook. The eGen COBOL Copybook Utility generates an XML document, an XML Schema, and a DTD from the employee information record. [Listing 2-2](#) shows the XML document generated from the employee record information, [Listing 2-3](#) shows the XML Schema, and [Listing 2-4](#) shows the DTD.

Listing 2-1 COBOL Copybook for Employee Information Record (emprec.cpy)

```
* -----
* emprec.cpy
* Employee record.
*
* @(#) $Id: emprec.cpy,v 1.2 1999/11/12 01:16:41 $
* -----
      02 emp-record.

      04 emp-ssn                                pic 9(9) comp-3.

      04 emp-name.
          06 emp-name-last                    pic x(15).
          06 emp-name-first                   pic x(15).
          06 emp-name-mi                      pic x.

      04 emp-addr.
          06 emp-addr-street                  pic x(30).
          06 emp-addr-st                     pic x(2).
          06 emp-addr-zip                    pic x(9).

* End
```

Listing 2-2 XML Document Generated from eGen COBOL Copybook Utility (emprec.xml)

```
<emprec>
  <empRecord>
    <empSsn>660337645</empSsn>
    <empName>
      <empNameLast>Purebred</empNameLast>
      <empNameFirst>Polly</empNameFirst>
      <empNameMi>P</empNameMi>
    </empName>
    <empAddr>
```

```

        <empAddrStreet>3235 Possum Park Ln.</empAddrStreet>
        <empAddrSt>TX</empAddrSt>
        <empAddrZip>758050000</empAddrZip>
    </empAddr>
</empRecord>
</emprec>

```

Listing 2-3 XML Schema Generated from eGen COBOL Copybook Utility (emprec.xsd)

```

<?xml version="1.0"?>
<schema
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <xsd:annotation>
    <xsd:documentation>
      Schema:      emprec
      Version:     1.0
      Source:      ../symbol/emprec.cpy
      Generated:   2000-09-27T19:19:42.857Z
      Created:     2000-09-27T19:19:43.708Z
      Modified:    1999-11-12T01:16:41.000Z
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="emprec">
    <xsd:complexType>

      <xsd:attribute name="date"
        type="xsd:dateTime"/>

      <xsd:attribute name="version"
        type="xsd:string"
        use="default"
        value="1.0"/>

      <xsd:element name="empRecord">
        <xsd:complexType>

          <xsd:element name="empSsn">
            <xsd:simpleType base="xsd:integer">
              <xsd:precision value="9"/>
              <xsd:minInclusive value="0">
            </xsd:simpleType>
            <!-- <%picture value="9(9)"/> -->
          </xsd:element>
        </xsd:complexType>
      </xsd:element>
    </xsd:complexType>
  </xsd:element>
</schema>

```

```
</xsd:element>

<xsd:element name="empName">
  <xsd:complexType>

    <xsd:element name="empNameLast"
      type="xsd:string"
      length="15"/>
    <!-- <%picture value="x(15)"/> -->

    <xsd:element name="empNameFirst"
      type="xsd:string"
      length="15"/>
    <!-- <%picture value="x(15)"/> -->

    <xsd:element name="empNameMi"
      type="xsd:string"
      length="1"/>
    <!-- <%picture value="x"/> -->

  </xsd:complexType>
</xsd:element> <!-- "empName" -->

<xsd:element name="empAddr">
  <xsd:complexType>

    <xsd:element name="empAddrStreet"
      type="xsd:string"
      length="30"/>
    <!-- <%picture value="x(30)"/> -->

    <xsd:element name="empAddrSt"
      type="xsd:string"
      length="2"/>
    <!-- <%picture value="x(2)"/> -->

    <xsd:element name="empAddrZip"
      type="xsd:string"
      length="9"/>
    <!-- <%picture value="x(9)"/> -->

  </xsd:complexType>
</xsd:element> <!-- "empAddr" -->

  </xsd:complexType>
</xsd:element> <!-- "empRecord" -->

</xsd:complexType>
```

```

    </xsd:element> <!--"emprec"-->

</schema>

```

Listing 2-4 DTD Generated from eGen COBOL Copybook Utility (emprec.dtd)

```

<!--
! DTD emprec 1.0
!
! Definition:    emprec
! Version:      1.0
! Source:       ../symbol/emprec.cpy
! Generated:    2000-09-27T19:18:25.084Z
! Created:      2000-09-27T19:18:24.937Z
! Modified:     1999-11-12T01:16:41.000Z
!-->

<!ELEMENT emprec
( empRecord )>

<!ATTLIST emprec
    date CDATA #DEFAULT "unknown">
    <!-- format="ccyy-mm-ddThh:mm:ss.mmmZ" -->

<!ATTLIST emprec
    version CDATA #DEFAULT "1.0">

<!-- empRecord -->
<!ELEMENT empRecord
( empSsn ,
  empName ,
  empAddr )>

<!-- empRecord.empSsn -->
<!ELEMENT empSsn
( #PCDATA )>

<!-- empRecord.empName -->
<!ELEMENT empName
( empNameLast ,
  empNameFirst ,
  empNameMi )>

<!-- empRecord.empName.empNameLast -->

```

```
<!-- ELEMENT empNameLast
      ( #PCDATA )>

<!-- empRecord.empName.empNameFirst -->
<!-- ELEMENT empNameFirst
      ( #PCDATA )>

<!-- empRecord.empName.empNameMi -->
<!-- ELEMENT empNameMi
      ( #PCDATA )>

<!-- empRecord.empAddr -->
<!-- ELEMENT empAddr
      ( empAddrStreet ,
        empAddrSt ,
        empAddrZip )>

<!-- empRecord.empAddr.empAddrStreet -->
<!-- ELEMENT empAddrStreet
      ( #PCDATA )>

<!-- empRecord.empAddr.empAddrSt -->
<!-- ELEMENT empAddrSt
      ( #PCDATA )>

<!-- empRecord.empAddr.empAddrZip -->
<!-- ELEMENT empAddrZip
      ( #PCDATA )>

<!-- End -->
```

3 Configuring the BEA WebLogic Java Adapter for Mainframe

BEA WebLogic Java Adapter for Mainframe (JAM) is configured with the JCRMGW configuration file. Configuration parameters define remote domains, the SNACRM, Local LU and SNA stack library, SNA links, and local and remote services.

The following topics provide instructions and information for configuring JAM:

- [General Steps for Defining the Gateway Configuration](#)
- [About the JCRMGW Configuration File](#)
- [JCRMGW Configuration File Sections](#)
- [Sample Configuration File](#)
- [Post Configuration Requirements](#)

General Steps for Defining the Gateway Configuration

Whether you are defining a new gateway configuration or modifying an existing one, both processes are similar. Defining a gateway configuration requires the following steps:

1. Determine which keyword information needs to be used to define the gateway as appropriate for your system configuration.
2. Create or edit the `jcrmgw.cfg` file with the gateway information.

Refer to the following sections for general information about the JCRMGW configuration file and detailed information about the keywords you need to define in your configuration file.

After you perform these steps, you are ready to start the WebLogic Server and verify that the gateway is ready. Refer to [Post Configuration Requirements](#) for more information about verifying that the gateway is ready.

About the JCRMGW Configuration File

The JAM Gateway, JCRMGW, uses the `jcrmgw.cfg` file to control much of its operation. The JCRMGW configuration file defines the SNACRM, stack (if required), links, and local and remote services that comprise the gateway environment. This file must be created and located in the WebLogic installation directory. A sample `jcrmgw.cfg` file is located in the JAM `examples/sample.jar` referred to in the *BEA WebLogic Java Adapter for Mainframe Installation Guide*.

The general format of the `jcrmgw.cfg` configuration file is as follows:

- The file is made up of six sections containing parameters. Lines beginning with an asterisk (*) indicate the beginning of a specific section. Each such line contains the name of the section immediately following the *. The following sections are defined:

- [JC_REMOTE_DOMAINS Section](#)
 - [JC_SNACRM Section](#)
 - [JC_SNASTACKS Section](#)
 - [JC_SNALINKS Section](#)
 - [JC_LOCAL_SERVICES Section](#)
 - [JC_REMOTE_SERVICES Section](#)
- Parameters are generally specified by: *KEYWORD* = *value*. This sets *KEYWORD* to *value*. Valid keywords are described in the following sections. *KEYWORDS* are reserved. They cannot be used as *values* unless they are quoted.
 - Input fields are separated by at least one space (or tab) character.
 - “#” introduces a comment. A new line ends a comment.
 - Blank lines and comments are ignored.
 - Comments can be freely attached to the end of any line.

JCRMGW Configuration File Sections

The following sections describe the significant keywords within specific sections of the `jcrmghw.cfg` file that define new gateway configurations.

JC_REMOTE_DOMAINS Section

This section provides an alias for associating mainframe applications with services and links. Any given entry in this section must be named. A name is provided by using a label on any one of the keywords that comprise the entry. Code examples will use a label on the first keyword of a given entry.

Valid Keywords

Following is a list of valid keywords for the `JC_REMOTE_DOMAINS` section:

Keyword	Required/ Optional	Description
DOMAINID	Required	Name of a partner system

Keyword Definitions

Following is more detailed information about the JC_REMOTE_DOMAINS section keyword:

DOMAINID=<string>

<string> is any name to be used for identifying a partner system. This keyword/value pair is required and has no default value.

Following is an example.

```
CRCICS1 DOMAINID="TestCICS"
```

JC_SNACRM Section

This section identifies the SNACRM that this gateway talks to. There is one SNACRM for each JCRMGW. Any given entry in this section must be named. A name is provided by using a label on any one of the keywords that comprise the entry. Code examples will use a label on the first keyword of a given entry.

Valid Keywords

Following is a list of valid keywords for the JC_SNACRM section:

Keyword	Required/ Optional	Description
GROUP	Required	Name of group correlating gateway with SNACRM
SNACRMADDR	Required	Symbolic TCP address

Keyword Definitions

Following is more detailed information about the JC_REMOTE_DOMAINS section keywords:

GROUP=<string>

<string> is any name to be used to correlate a gateway with a SNACRM. The name is used as part of the file name for the SNACRM logs. In the case of a SNACRM that is started independently of the gateway, this name must match the group name used on the SNACRM command line, even if the default name is used. The keyword/value pair is optional and has a default value of "SNAGROUP".

The following listing contains an example.

```
GROUP="CRAUTH"
```

The SNACRM expects a gateway signon for group CRAUTH.

SNACRMADDR=<string>

<string> is a symbolic TCP address in the form of: "//hostname:port"

Host name is the name of the machine that runs the SNACRM. Port is an unused decimal port number that the SNACRM uses to talk to the Java gateway. In the case of a SNACRM that is started independently of the gateway, this address must match the address used on the SNACRM command line. When the gateway is started, it tries to contact the SNACRM at the address.

The following listing contains an example.

```
MYSNACRM SNACRMADDR="//dalhp55:6942"
```

The gateway will look for a SNACRM on a machine named dalhp55 at port 6942.

JC_SNASTACKS Section

This section identifies the Local LU used for the SNACRM along with the stack identifier for the SNA stack library to be used. Only one local LU and one stack can be specified for a SNACRM. Any given entry in this section must be named. A name is provided by using a label on any one of the keywords that comprise the entry. Code examples will use a label on the first keyword of a given entry.

Valid Keywords

Following is a list of valid keywords for the JC_SNASTACKS section:

Keyword	Required/ Optional	Description
LOCALLU	Required	Alias of local LU to be used with SNACRM
STACKTYPE	Required	Name of SNA stack support library to load

Keyword Definitions

Following is more detailed information about the JC_SNASTACKS section keywords:

LOCALLU=<string>

<string> is an alias for the local LU to be used by the SNACRM. This must match a corresponding LU alias defined in the SNA Stack configuration. This alias may or may not match the actual LU name. This keyword/value pair is required and has no default value.

The following listing contains an example.

```
LOCALLU="AUTHAPP"
```

The SNACRM tries to use the local LU which has been defined in the SNA stack configuration with an alias of AUTHAPP.

STACKTYPE=[hp60 | ibm50 | ibm60 | ms40 | spx60 | sun91]

One of the specified tokens must be used. These names determine which SNA stack support library will be loaded.

The following stacks can be identified:

- hp60 = Hewlett Packard SNA 2 Plus 6.0 on HP-UX
- ibm50 = IBM Communications server 5.0 on WINNT or AIX
- ibm60 = IBM Communications server 6.0 on WINNT
- ms40 = Microsoft SNA Server 4.0 SP3 on WINNT
- spx60 = Data Connection Snapix 6.0 on Solaris
- sun91 = Sunlink 9.1 on Solaris

The keyword/value pair is required and has no default value.

The following listing contains an example.

```
ibmcsaix STACKTYPE="ibm50"
```

The SNACRM tries to load the library for IBM Communication Server 5.0.

JC_SNALINKS Section

This section identifies partner mainframe application regions. Multiple links for a single SNACRM are supported. Any given entry in this section must be named. A name is provided by using a label on any one of the keywords that comprise the entry. Code examples will use a label on the first keyword of a given entry.

Valid Keywords

Following is a list of valid keywords for the JC_SNALINKS section:

Keyword	Required/ Optional	Description
MAXSESS	Optional	Maximum number of SNA sessions that can be started for a link
MINWIN	Optional	Minimum number of SNA sessions that will be contention winners
MODENAME	Required	Name of mode definition
RDOM	Required	Name of remote domain

Keyword	Required/ Optional	Description
RLUNAME	Required	Alias for remote LU
SECURITY	Optional	Indicates level of security

Keyword Definitions

Following is more detailed information about the JC_SNALINKS section keywords:

MAXSESS=*nn* (4)

nn is the maximum number of SNA sessions that can be started for this link. The actual value used is negotiated with the partner and can be lower than this value if the partner is configured with a lower value. This value includes two sessions for the SNA Service manager mode. The lowest usable value is 4; it provides two sessions for the application and two sessions for the service manager. This keyword/value pair is optional and has a default value of 4.

The following listing contains an example.

```
MAXSESS=16
```

The maximum number of sessions on this link is set to 16 for all modes combined.

MINWIN=*nn* (MAXSESS/2)

nn is the minimum number of SNA sessions that will be contention winners for the SNACRM. This keyword is not required and defaults to one half the number of MAXSESS. In most cases this is suitable unless asymmetric winners are desired due to application requirements. Also, the default value may not be appropriate if the maximum is negotiated down to a value lower than half of the specified maximum. This value includes one session for the SNA Service Manager mode. In general, the lowest practical value is 2, which provides one session for the application and one session for the service manager. See IBM's SNA documentation for a complete discussion of session limits and contention winners. The keyword/value pair is optional and has a default value of half the value of MAXSESS.

The following listing contains an example.

```
MINWIN=8
```

The number of contention winner sessions on this link is set to 8 for all modes combined.

MODENAME=<string>

<string> is the name of a mode definition to be used for applications on this link. This must match a corresponding mode name defined in the SNA Stack configuration and the VTAM mode table. A valid mode name is provided by mainframe support personnel. This keyword/value pair is required and has no default value.

The following listing contains an example

```
MODENAME="SMSNA100"
```

The SNACRM will use the SMSNA100 mode for applications on this link.

RDOM=<string>

<string> is a name previously defined as a remote domain. The remote domain naming is a mechanism for grouping links. This keyword/value pair is required and has no default value.

The following listing contains an example.

```
RDOM="CRCICS1"
```

This link is associated with the remote domain named CRCICS1.

RLUNAME=<string>

<string> is an alias for the remote LU representing the partner application to be used by the SNACRM. This must match a corresponding partner/remote LU alias defined in the SNA Stack configuration. This alias may or may not match the actual remote LU name. This keyword/value pair is required and has no default value.

The following listing contains an example.

```
RLUNAME="AUTHAPPL"
```

The SNACRM routes all traffic for this link to a remote application defined with an alias of AUTHAPPL. This alias must be defined in the SNA stack configuration for a valid mainframe application name.

SECURITY=[LOCAL | IDENTITY | VERIFY] (LOCAL)

This is an optional keyword. If entered, one of the specified tokens must be used, and the mainframe connection be configured to match.

The meaning of the values are:

LOCAL = All security is handled by the local system and the link itself has no security requirement.

IDENTIFY = A userid will be passed on to the mainframe. This userid can originate with the client application or it can be a default userid supplied with the `-u` option on the gateway startup.

VERIFY = A userid and password will be passed onto the mainframe. The userid can originate with the client application or it can be a default userid supplied with the `-u` option on the gateway startup. The password must be supplied by the client application.

This keyword/value pair is optional and has a default value of LOCAL.

The following listing contains an example.

```
SECURITY=IDENTIFY
```

A userid is required for all requests made on this link. Note that the mainframe configuration for the remote LU (*i.e.*, a CICS connection definition) must have a security level that matches.

JC_LOCAL_SERVICES Section

This section maps incoming mainframe program names to a Home interface for an EJB that will service the request. Any given entry in this section must be named. A name is provided by using a label on any one of the keywords that comprise the entry. Code examples will use a label on the first keyword of a given entry.

Valid Keywords

Following is a list of valid keywords for the JC_LOCAL_SERVICES section:

Keyword	Required/ Optional	Description
RNAME	Required	Name of remote resource associated with a service

Keyword	Required/ Optional	Description
SCHEMA	Required for use with WebLogic Process Integrator	Identifies the generated dataview.

Keyword Definitions

Following is more detailed information about the `JC_LOCAL_SERVICES` section keyword:

RNAME=<RRRRRRR>

RNAME is the remote resource name associated with this service. For a CICS Distributed Program Link, this is the actual program name that was invoked from the mainframe. For a DTP style request, the resource name must conform to the TP ID requirements of the originating system. A CICS DPL Program and an IMS Transaction ID are limited to eight characters.

The label on this entry must be the name of a valid home interface for an Enterprise Java Bean registered with JNDI. This keyword/value pair is required and has no default value.

The following listing contains an example.

```
StatelessSession.TraderHome RNAME="DPL1SVR"
```

DPL1SVR is the name of the program that was invoked from the mainframe and `StatelessSessions.TraderHome` is the name of the home interface which will be used to invoke the EJB that will service this request.

SCHEMA=<string>

This keyword is used to identify the generated dataview for use in decoding or encoding data streams. SCHEMA is used by the `JAMTOJMS` or `AppView` EJB's when integrating with WebLogic Process Integrator.

JC_REMOTE_SERVICES Section

This section maps remote mainframe program names to method names that can be used by a local application to invoke the remote request. Any given entry in this section must be named. A name is provided by using a label on any one of the keywords that comprise the entry. Code examples will use a label on the first keyword of a given entry.

Valid Keywords

Following is a list of valid keywords for the JC_REMOTE_SERVICES section:

Keyword	Required/ Optional	Description
FUNCTION	Optional	Method of request invocation accepted by the mainframe
RDOM	Required	Name of remote domain
RNAME	Required	Remote resource name associated with a service
SCHEMA	Required for use with WebLogic Process Integrator	Identifies the generated dataview.
TRANTIME	Optional	Maximum number of milliseconds client will allow for a host request to respond

Keyword Definitions

Following is more detailed information about the JC_REMOTE_SERVICES section keywords:

FUNCTION=[DTP | DPL] (DPL)

This is an optional keyword used to indicate the method of request invocation accepted by the mainframe. DPL is the default and is only valid when used with a CICS partner and an application that can be invoked using a Distributed Program Link. DTP should be used when invoking services from an IMS server.

This keyword/value pair is optional and has a default value of DPL.

The following listing contains an example.

```
FUNCTION=DTP
```

This remote service will be invoked as an IMS application rather than a program link.

RDOM=<string>

<string> is a name previously defined as a remote domain. This keyword/value pair is required and has no default value.

The following listing contains an example

```
RDOM="CRCICS1"
```

This service will be associated with the remote domain named CRCICS1.

RNAME=<[MMMM:]RRRRRRRR>

Rname is the remote resource name associated with this service. For a CICS Distributed Program Link, this is the actual program name to be invoked. The first portion of this value (MMMM:) is optional and can be an alternate mirror transaction identifier. An alternate mirror transaction is useful for some mainframe database deployments as well as security or charge back systems. The alternate mirror transaction cannot exceed 4 characters in length. The second portion (RRRRRRRR) is the program name to invoke for a CICS DPL. For a DTP style request, the resource name must conform to the Tran ID requirements of IMS.

The following listing contains an example.

```
RNAME="AUTH:DPLQRY"
```

This Rname is a program named DPLQRY and will use an alternate mirror transaction name of AUTH.

SCHEMA=<string>

This keyword is used to identify the generated dataview for use in decoding or encoding data streams. SCHEMA is used by the JAMTOJMS or AppView EJB's when integrating with WebLogic Process Integrator.

TRANTIME=nnn (30000)

nnn is the maximum number of milliseconds the client application will block before a host request is timed out. This keyword/value pair is optional and has a default value of 30000.

The following listing contains an example.

```
TRANTIME=120000.
```

This remote service will time out if the mainframe does not respond within 2 minutes (120 seconds).

Sample Configuration File

The following example illustrates a basic jcrmgw.cfg file.

Listing 3-1 Sample jcrmgw.cfg Configuration File

```
*JC_REMOTE_DOMAINS
#
CICS410          DOMAINID="410"
#
*JC_SNACRM
#
SNACRMAN         SNACRMADDR="//da1nt66:8650"
                  GROUP="SNAG1"

*JC_SNASTACKS
#
IBMCSAIX         STACKTYPE="IBM50"
                  LOCALLU="LUNT66B"
#
*JC_SNALINKS
#
C41XB01          RLUNAME="C410XB01"
```

```

                                RDOM="CICS410"
                                MODENAME="SMSNA100"
                                MAXSESS=8
                                MINWIN=6
#
*JC_LOCAL_SERVICES
#
TraderHome      RNAME="DPL1SVR"
#
JC_REMOTE_SERVICES
DPLINIT         RDOM="CICS410"
                                RNAME="PRIM:DPLINIT"
                                TRANTIME=10000
TOUPPER         RDOM="CICS410"
                                RNAME="TOUPDPLS"
                                TRANTIME=10000
demoRead        RDOM="CICS410"
                                RNAME="DPLDEMOR"
                                TRANTIME=10000
demoUpdate      RDOM="CICS410"
                                RNAME="DPLDEMOU"
                                TRANTIME=10000
demoCreate      RDOM="CICS410"
                                RNAME="DPLDEMOC"
                                TRANTIME=10000
demoDelete      RDOM="CICS410"
                                RNAME="DPLEMOD"
                                TRANTIME=10000

imsInsert       RDOM="IMS51"
                                FUNCTION=DTP
                                RNAME="IMSSVR12"

```

Post Configuration Requirements

Start WebLogic Server and verify that the *SnacTask Startup Complete* message indicates the gateway is ready. Also you can verify that the CICS connection is acquired using the CICS CEMT I CON (*) Command to verify your connection is in the ACQUIRED state, indicating the link is ready for use.

4 Developing Java Applications

The BEA WebLogic Java Adapter for Mainframe (JAM) code generation tool, eGen COBOL, produces Java code from COBOL copybook source files. This generated Java code consists of both data mapping code and framework code for any application that will invoke the mainframe application through the gateway.

This section describes how the base Java application is generated, what role the COBOL copybook plays in the generation, and how to determine the type of application you want to generate.

This section discusses the following topics:

- [Building the Base Java Application](#)
- [Obtaining the COBOL Copybook](#)
- [Generating the Java Application Source](#)
- [Using Client Diagnostic Features](#)

Building the Base Java Application

The JAM code generation tool allows you to produce working Java code that functions within a Java Server execution model. The Java code produced is centered around typed data buffers.

In COBOL/CICS terms, a typed data buffer represents a group data item, or record, defined by a COBOL copybook that specifies the COMMAREA data area used by a CICS DPL program.

In Java terms, a typed data buffer is an object (instantiation) of a class type. The fields of the class correspond to the data fields within a record. Field values are retrieved and modified through the use of accessor functions (such as get and set functions).

Dataview Concepts

Typed data buffers are handled by the Java `DataView` class, which performs all necessary conversions and translations of the data fields within each record. The result is the ability to send and receive data buffers between Java clients and COBOL/CICS DPL programs. All underlying data conversions are performed transparently to each side of the application; the Java side of the application manipulates the data buffers as Java objects, and the COBOL/CICS side manipulates them as COMMAREA linkage section data items.

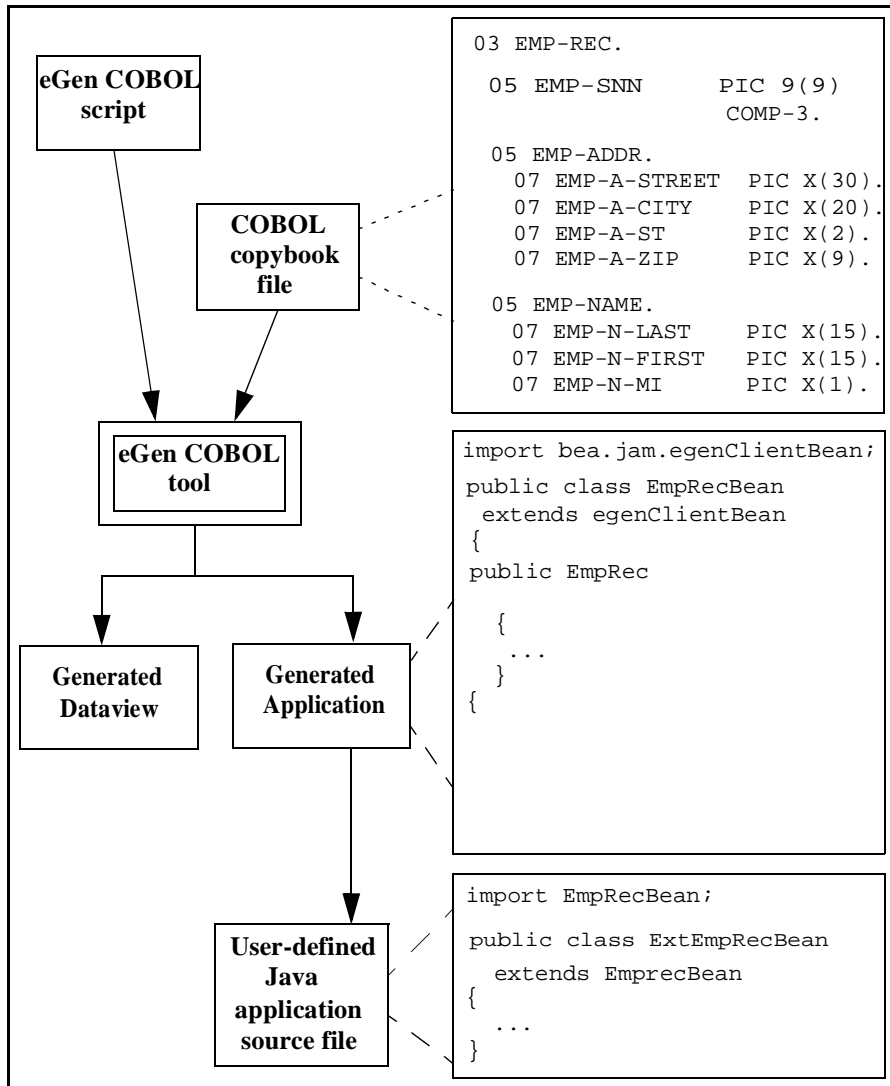
These Java files are then used by application programmers to write other Java source files that extend the classes defined in the generated source files. The user-defined classes typically add methods and variables that embody the business logic of the application.

The generated source files provide a simple framework centered around the typed data buffers (i.e., the COMMAREAs) of the CICS application. It is up to the Java application programmer to extend this framework of classes.

The heart of a JAM application is the sending and receiving of data records between the Java application system and the remote (mainframe) system. This data transfer is accomplished by employing a `DataView` that corresponds to a mainframe data record. On the mainframe side, the `DataView` represents the CICS COMMAREA data that is sent between DPL programs. On the Java side, the `DataView` represents a class object containing the COMMAREA data fields. The JAM application framework provides all of the data conversions necessary to send and receive data records between Java systems and the mainframe.

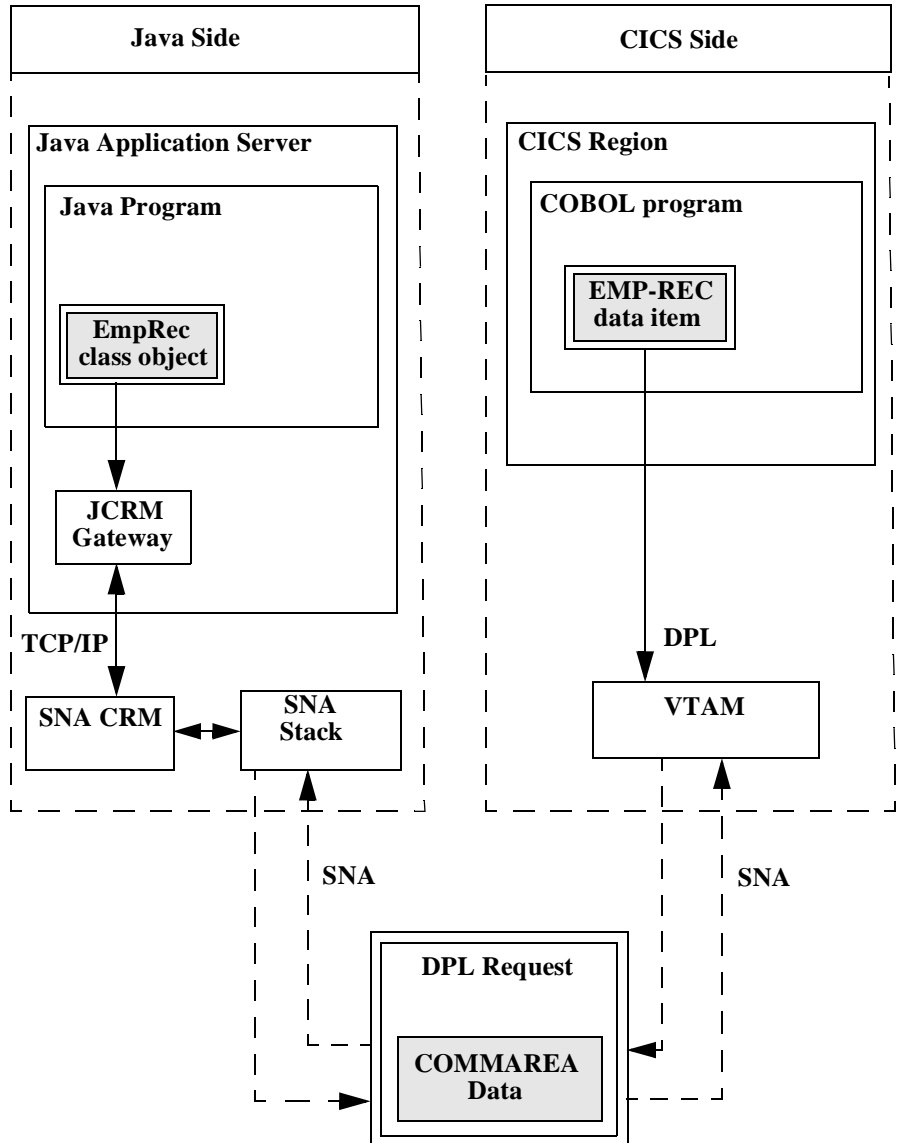
The following figure provides an illustration of how the underlying JAM generation tools work together.

Figure 4-1 Understanding the Underlying JAM Tools



The following figure provides an illustration of the JAM general application transport method.

Figure 4-2 JAM General Application Transport Method



Obtaining the COBOL Copybook

A mainframe COBOL/CICS, IMS, or mainframe application typically uses a copybook source file to define its COMMAREA. This file is specified in a COPY directive within the LINKAGE SECTION of the source program.

If the CICS application does not use a copybook file (but simply defines the COMMAREA directly in the program source), you will have to create one from the definition contained in the program source.

The JAM code generation tools support most of the COBOL data types and data clauses, however, some obsolete constructs along with floating point data types are not supported. Also, a few COBOL features are not supported. For unsupported constructs, you will have to determine if the copybook can be used without modification. In most cases, unsupported items are treated as alphanumeric data types or are simply ignored.

Each copybook's contents (which define a COMMAREA record) are parsed by the eGen COBOL application generator tool, producing DataView sub-classes that provide facilities to:

- Convert COBOL data types to and from Java data types. This includes conversions for mainframe data formats and code pages.
- Convert COBOL data structures to and from Java data structures.
- Provide tools that may be used to convert the provided data structures into other arbitrary formats.

Creating a New Copybook

If you are producing a new application on the mainframe or modifying one that did not previously support DPL, then one or more new copybooks are required. You should keep in mind the COBOL features and data types supported by JAM as you create these copybooks.

Using an Existing Copybook

When a mainframe application has an existing DPL interface, it most likely has the data for that interface described in a COBOL copybook. The first thing that must be done is to verify that no COBOL features or data types that JAM does not support are used in the interface. The easiest way to check a copybook is to attempt to process it.

An example COBOL copybook source file is shown in the following listing. It is assumed that the copybook is identical to the copybook used by the mainframe COBOL application programs.

Note: Some of the code sample listings in this topic have field names in bold for easier reading. Also, Comment-numbered items have corresponding comments at the bottom of each script example.

Listing 4-1 Sample `emprec.cpy` COBOL Copybook

```
1      *-----
2      * emprec.cpy
3      *      An employee record.
4      *-----
5
6      02      emp-record. (Comment 1)
7
8              04      emp-ssn                      pic 9(9)  comp-3.
9
10             04      emp-name.
11                     06      emp-name-last      pic x(15). (Comment 2)
12                     06      emp-name-first      pic x(15).
13                     06      emp-name-mi         pic x.
14
15             04      emp-addr. (Comment 3)
16                     06      emp-addr-street      pic x(30).
17                     06      emp-addr-st          pic x(2).
18                     06      emp-addr-zip         pic x(9).
19
20      * End
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-1 Script Comments

Comment 1	Declaration of a record (group) data item.
Comment 2	An elementary item.
Comment 3	An aggregate item.

The `emprec.cpy` copybook shown in the example declares a group item named `emp-record`. (This copybook is available in the `examples/samples.jar` file in JAM installation directory.

The JAM eGen Cobol utility parses a COBOL copybook and generates a Java class that encapsulates the data record declared in the copybook. It does this by parsing an eGen script file containing a `DataView` definition similar to the following example.

Listing 4-2 Sample `emprec.egen` Script

```
generate view Sample.EmployeeRecord from emprec.cpy
```

The prior example specifies that the COBOL copybook file named `emprec.cpy` is to be parsed, and that a Java source file named `EmployeeRecord.java`, containing the definition of class `EmployeeRecord` in package `Sample`, is to be generated from it.

Assuming this script was saved into a file named `emprec.egen`, the following shell command parses the copybook file named `emprec.cpy` and attempts to generate a Java source file named `EmployeeRecord.java` in the current directory:

Listing 4-3 Sample Copybook Parse Command

```
egencobol emprec.egen
```

If no error or warning messages are issued, the copybook is compatible with JAM and the source is created.

Note: You must establish both an execution path that includes the JAM distribution bin directory and a Java class path that refers to the JAM distribution jar file for this command to work properly. Refer to the “Error Messages” section of this document for suggestions on resolving other problems.

The following example illustrates the resulting generated Java source file, `EmployeeRecord.java` with unimportant comments and implementation details removed for clarity.

Listing 4-4 `Generated EmployeeRecord.java Source File`

```
//EmployeeRecord.java
//Dataview class generated by egencobol emprec.cpy

package Sample;(Comment 1)

//Imports

import bea.dmd.DataView.DataView;
...

/**DataView class for EmployeeRecord buffers*/

public final class EmployeeRecord (Comment 2)
    extends DataView
{
    ...

    // Code for field "emp-ssn"
    private BigDecimal    m_empSsn;(Comment 3)

    public BigDecimal    getEmpSsn() {...}(Comment 4)

    /** DataView subclass for emp-name Group */
    public final class EmpNameV (Comment 5)
        extends DataView
    {
        ...

        // Code for field "emp-name-last"
        private String    m_empNameLast;

        public void setEmpNameLast(String value) {...}
        public String getEmpNameLast() {...}(Comment 6)
```

```
// Code for field "emp-name-first"
private String m_empNameFirsrt;

public void setEmpnameFisrt (String value) {...}
public String getEmpNameFirst() {...}

// Code for field "emp-name-mi"
private String m_empNameMi;

public void setEmpNameMi (String value) {...}
public String getEmpnameMi() {...}
}

// Code for field "emp-name"
private EmpNameV m_empname; (Comment 7)

public EmpnameV getEmpname() {...}

/**DataView subclass for emp-addr Group */
public final class EmpAddrV
    extends DataView
{
    ...

    // Code for field "emp-addr-street"
    private String m_empAddrStreet;

    public void setEmpAddrStreet(Street value) {...}
    public String getEmpAddrStreet() {...}

    // Code for field "emp-addr-st"
    private String m_empAddrSt;

    public void setEmpAddrSt(String value) {...}
    public String getEmpAddrSt() {...}

    // Code for field "emp-addr-zip"
    private String m_empAddrZip;

    public void setEmpAddrZip(String value) {...}
    public String getEmpAddrZip() {...}
}

// Code for field "emp-addr"
private EmpAddrV m_empAddr;

public EmpAddrV getEmpAddr() {...}
}

//End EmployeeRecord.java
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-2 Script Comments

Comment 1	The package name is defined in the eGen script.
Comment 2	The data record is encapsulated in a class that extends the DataView class.
Comment 3	Each class member variable corresponds to a field in the data record.
Comment 4	Each data field has accessor functions.
Comment 5	Each aggregate data field has a corresponding nested inner class that extends the DataView class.
Comment 6	Each data field within an aggregate data field has accessor functions.
Comment 7	Each COBOL data field name is converted into a Java identifier.

Additional options may be specified in the eGen script to change details of the DataView generation. For example, the following script will generate a dataview class that uses codepage cp500 for conversions to and from mainframe format. If the codepage clause is not specified the default codepage of cp037 is used.

```
generate view Sample.EmployeeRecord from emprec.cpy codepage cp500
```

The following script will generate additional output intended to support use of the dataview class with XML data.

```
generate view Sample.EmployeeRecord from emprec.cpy support xml
```

Additional files are listed in Table 4-3.

Table 4-3 Additional Files for DataView XML Support.

File Name	File Purpose
<i>classname.dtd</i>	XML DTD for XML messages accepted and produced from this dataview.

File Name	File Purpose
<i>classname.xsd</i>	XML schema for XML messages accepted and produced from this dataview.
<i>classnameHelper.java</i>	Source code for XML helper class. This class may be used when communicating with JAM from WebLogic Process Integrator. This class will reside in the same package as the dataview class.

Generating the Java Application Source

In addition to the COBOL copybook requirement, you must also determine which of the following application models you want to generate.

- [Generating a Servlet-Only JAM Application](#)
- [Generating a Client Enterprise Java Bean-Based Application](#)
- [Generating a Server Enterprise Java Bean-Based Application](#)
- [Generating a Stand-Alone Client Application](#)

The one you choose depends on the type of application you are building.

For all of the following applications you can generate, you must provide a script file containing definitions for the application, including the COBOL copybook file name, the DataView class names, and so forth.

Generating a Servlet-Only JAM Application

This type of application produces a Java servlet application that executes within a Java server environment (such as WebLogic Server). It is started from a web browser when the user enters a URL that is configured to invoke the servlet. The servlet presents an HTML form containing data fields and buttons. The buttons can be configured to invoke:

- EJB methods

- Remote gateway services (via the JCRMGW)

In general, servlets generated by eGen COBOL are intended for testing purposes and are not easily customized to provide a more aesthetically pleasing interface.

Generating a Servlet-only JAM application consists of the following steps;

1. Creating a script.
2. Processing a script.
3. Working with the generated files.
4. Customizing the application.

The following topics discuss these steps and offer examples of each.

Creating a Script

In order to produce a servlet-only application, the script file that describes your DataViews must be enhanced to describe the mainframe services accessed, the browser pages produced, and the servlets that produce them. Service definitions look like the following listing.

Listing 4-5 Sample Service Definition

```
service getSalary
    accepts EmployeeRecord
    returns EmployeeRecord
```

This listing defines a service named `getSalary` that accepts an input buffer of type `EmployeeRecord` and returns an output buffer of type `EmployeeRecord`. It is this service name which also requires an entry in the `jcrmgw.cfg` file.

A browser page that uses this service might be defined as the following.

Listing 4-6 Sample Page Definition

```
page EmployeeQuery "Employee Salary Query"
{
```

```
view EmployeeRecord

buttons
{
    Query service (getSalary)
        shows EmployeeQuery
}
}
```

This listing defines an HTML page named `EmployeeQuery`, with a text title of “Employee Salary Query”, that displays an `EmployeeRecord` record object as an HTML form. It also specifies that the form has a button labeled “Query.” When the button is pressed, the service `getSalary` is invoked and is passed the contents of the browser page as an `EmployeeRecord` object (the fields of which may have been modified by the user). Afterwards, the `EmployeeQuery` page is used to display the results.

The servlet that serves this page might be defined like the following listing.

Listing 4-7 Sample servlet Definition

```
servlet EmployeeQueryServlet shows EmployeeQuery
```

This listing defines an application servlet class named `EmployeeQueryServlet`, and specifies that it displays the HTML page named `EmployeeQuery` as its initial display page.

The following listing shows a complete script for defining a servlet application.

Listing 4-8 Sample Servlet-Only Script

```
1 #-----
2 # empServlet.egen
3 #   JAM script for a servlet-only application.
4 #
5 #   $Id: empServlet.egen,v 1.2 2000/01/25 18:34:14 david Exp$
6 #-----
7
```

```
8  # DataViews (typed data records)
9
10 view sample.EmployeeRecord (Comment 1)
11     from emprec.cpy support xml
12
13  # Services
14
15  service sampleCreate (Comment 2)
16      accepts EmployeeRecord
17      returns EmployeeRecord
18
19  service sampleRead (Comment 2)
20      accepts EmployeeRecord
21      returns EmployeeRecord
22
23  service sampleUpdate (Comment 2)
24      accepts EmployeeRecord
25      returns EmployeeRecord
26
27  service sampleDelete (Comment 2)
28      accepts EmployeeRecord
29      returns EmployeeRecord
30
31  # Servlet HTML pages
32
33  page initial "Initial page" (Comment 3)
34  {
35      view EmployeeRecord (Comment 4)
36
37      buttons
38      {
39          "Create" (Comment 5)
40              service ("sampleCreate")
41              shows fullPage
42
43          "Read" (Comment 5)
44              service ("sampleRead")
45              shows fullPage
46      }
47  }
48
49  page fullPage "Complete page"
50  {
51      view EmployeeRecord
52
53      buttons
54      {
55          "Create"
56              service ("sampleCreate")
```

```

57             shows fullPage
58
59         "Read"
60             service ("sampleread")
61             shows fullpage
62
63         "Update"
64             service ("sampleDelete")
65             shows fullpage
66
67         "Delete"
68             service ("sampleDelete")
69             shows fullpage
70     }
71 }
72
73 # Servlets
74
75 servlet sample.SampleServlet (Comment 6)
76     shows initial
77
78 # End

```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-4 Script Comments

Comment 1	Defines a DataView class, specifying its corresponding copybook source file and its package file.
Comment 2	Defines a service function and its input and output parameter types.
Comment 3	Defines an HTML page to be displayed by the servlet.
Comment 4	Specifies the DataView class to display on the page.
Comment 5	Defines a button and its associated class method.
Comment 6	Defines a servlet class and its initial HTML display page.

Processing a Script

To process the script, issue the command in Listing 4-9. The `egencobol` command involves the JVM and is equivalent to `java com.bea.jam.egen.EgenCobol empServlet.egen`.

Listing 4-9 Sample Script Process Command

```
egencobol empServlet.egen
emprec.cpy, Lines: 21 Errors: 0, Warnings:0
Generating sample.EmployeeRecord...
Generating group emp-name
Generating group emp-addr
Generating sample.SampleServlet...
```

Generated Files

This script command generates the following files.

Table 4-5 Sample Script Generated Files

Files	Content
SampleServlet.java	Servlet source code
EmployeeRecord.java	Source for the DataView object
EmployeeRecord.dtd	Generated DTD
EmployeeRecord.xsd	Generated XML/Schema
EmployeeRecordHelper.java	Helper class source code

SampleServlet.java Source File

The following listing illustrates the contents of the generated `SampleServlet.java` source file (with some parts omitted).

Listing 4-10 Sample SampleServlet.java Contents

```
// SampleServlet.java
//
// Servlet class generated by eGencobol on 25-Jan-2000.

package Sample;

// Imports

import javax.servlet.http.HttpServlet;
import weblogic.html.ServletPage;
import com.bea.dmd.DataView.DataView;
import com.bea.jam.egen.EgenServlet;
...

/** servlet class for EmployeeRecord buffers. */
public class SampleServlet
    extends EgenServlet
{
    /** Create a new servlet. */
    public SampleServlet()
    {
        ...
    }

    /** Get an instance of the initial DataView for this
    Servlet.*/
    protected DataView initialDataView()
    {
        ...
    }

    ...
}

//End SampleServlet.java
```

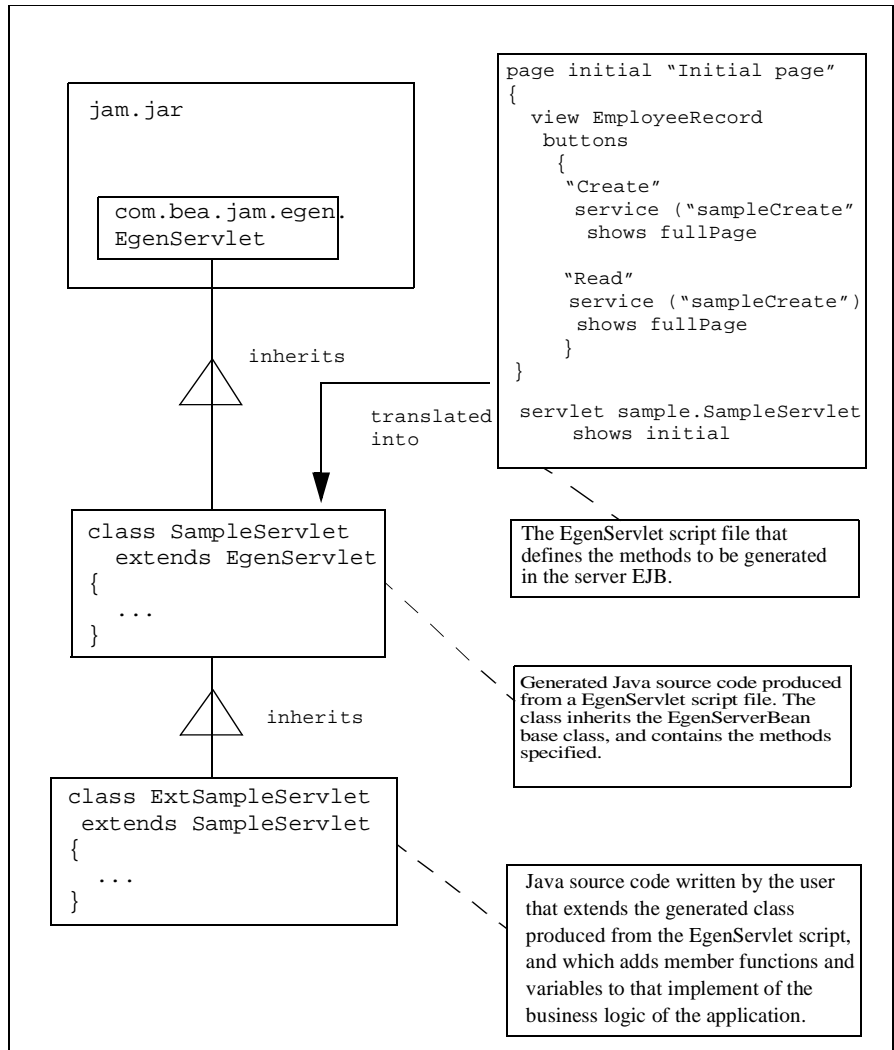
Customizing a Servlet-Only JAM Application

The generated Java classes produced for servlet applications are intended for proof of concept and prototypes, and can be customized in limited ways. It is presumed that some other development tool will be used to develop a servlet or other user interface on top of the generated EJBs or client classes.

This section describes the way that generated servlet code can be customized.

The following figure illustrates the relationships and inheritance hierarchy between the JAM classes comprising the application.

Figure 4-3 The JAM Servlet Class Hierarchy



The generated Java code for a servlet application Sample Servlet is a class that inherits class EgenServlet. Class EgenServlet is provided in the JAM distribution jar file.

The base class illustrated in the following listing provides the basic framework for a servlet.

Listing 4-11 EgenServlet.java **Base Class**

```
//=====
// EgenServlet.java
//      The base class for generated servlets.
//=====

package bea.jam.egen;

//Imports

...

/*****
 * The base class for generated servlets
 */

abstract public class EgenServlet
    extends HttpServlet
{
    /** Perform an HTTP Get operation. */
    public void doGet(HttpServletRequest req,HttpServletResponse
resp)
        throws ServletException, IOException
    {
        DataView dv;
        HttpSession session=req.getSession(true);

        ...

        // Get the initial DataView data record
        dv = initialDataView();

        // Invoke the user-defined callback
        dv = doGetSetup(dv,session);

        // Convert the DataView into an HTML form
        ...
    }

    /** Perform a HTTP Post operation. */
    public void doPost(HttpServletRequest req,HttpServletResponse
```

```
resp)
    throws ServletException, IOException
{
    DataView dv;
    HttpSession session=reqgetSession(true);

    // Move the HTML form data into a DataView
    ...

    // Invoke the user-defined callback
    dv = doPostSetup(dv, session);

    // Execute the form button
    ...

    //Invoke the user-defined callback
    dv = doPostFinal(dv, session);

    // Convert the DataView into an HTML form
    ...
}

/** User exit for pre-presentation processing for a GET request.
 */
public DataView doGetSetup (DataView in, HttpSession session)
{
    // Default behavior may be overridden
    return in;
}

/**User exit for before business logic processing for a POST
request. */
public DataView doPostSetup (DataView in, HttpSession session)
{
    // Default behavior, may be overridden
    return in;
}

/** User exit for after business logic processing for a POST
request. */
public DataView doPostFinal (DataView in, HttpSession session)
{
    // Default behavior, may be overridden
    return in;
}

/** Get an instance of the initial DataView for this servlet. */
protected abstract DataView initialDataView();

/**
 * The title for the initial page.
```

```
        * This should be initialized in the subclass constructor.
        */
        protected String  m_initialTitle;

    /**
     * The buttons for the initial page.
     * This should be initialized in the subclass constructor.
     */
    protected Button[] m_initialButtons;
}

// End EgenServlet.java
```

The `EgenServlet` base class provides functions for the GET and POST operations for the servlet's HTML page.

Both of these operations invoke the following default callback functions:

- `doGetSetup()` - invoked before the GET operation.

This function occurs prior to the presentation of the HTML page to the user's browser. Any changes made to the `DataView` object will be reflected in the contents of the HTML page.

- `doPostSetup()` - invoked before the POST operation.

This function occurs after the HTML page is presented and the user activates a form button. The `DataView` is sent to the `doPostSetup()` function, which operates on its contents. For example, validating the contents of the fields.

- `doPostFinal()` - invoked after the POST operation.

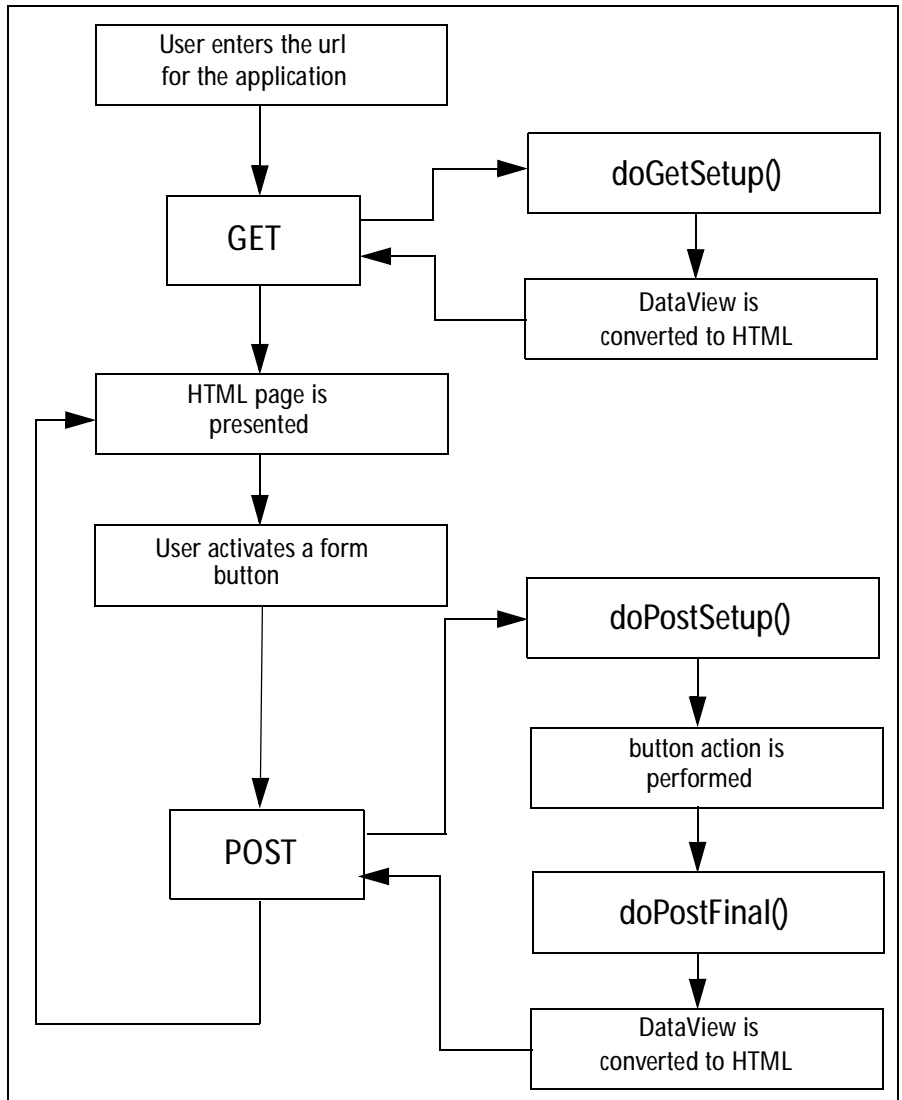
This function occurs prior to the presentation of the HTML page to the user's browser after activating a form button. Any changes made to the `DataView` object will be reflected in the contents of the HTML page.

Your class (`ExtSampleServlet.java`), which indirectly extends the `EgenServlet` base class, overrides these functions and provides additional business logic to modify the contents of the `DataView`. Each of these functions is passed to the `DataView` object containing the current record data. Each is expected to return a (potentially modified) `DataView` object.

Note: The overriding functions must have exactly the same signature as the functions in the base class.

The following illustration shows the sequence of operations that occur during the course of a user's browser session. For example, the series of events that occur within the EgenServlet class.

Figure 4-4 User Browser-Session Flowchart



Example ExtSampleServlet.java Class

The following listing shows an sample ExtSampleServlet class that extends the generated SampleServlet class, and adds a validation function (isSsnValid()) for the emp-ssn (m_empSsn) field of the DataView EmployeeRecord class. The three callback functions are overridden by the functions in the extended class. If the emp-ssn field is determined to be invalid, an exception is thrown.

Exceptions are caught by the Java server (BEA WebLogic Server) and cause a simple informational text page to be presented to the user's browser. Any string text associated with the exception is displayed along with an execution stack trace that was in effect at the time that the exception was thrown.

Listing 4-12 Sample ExtSampleServlet.java Contents

```
//=====
// ExtSampleServlet.java
// Example class that extends a generated JAM servlet application.
//=====

package Sample;

// System imports

import java.math.BigDecimal;

import javax.servlet.http.HttpSession;
import com.bea.dmd.DataView.DataView;
import com.bea.jam.egen.EgenServlet;

// Local Imports

import sample.EmployeeRecord;
import sample.SampleServlet;

/*****
 * Extends the SampleServlet class, adding additional business
 * logic
 */

public class ExtSampleServlet
    extends SampleServlet
{
    // Public functions
```

```

/*****
 * User exit for pre-presentation processing for a GET
 * request. This is called prior to the presentation of the
 * first HTML page to the user's browser.
 */

public DataView doGetSetup (DataView in, HttpSession
session)
{
    EmployeeRecord erec;

    // Overrides the default behavior

    // Load default data into the empty DataView
    erec = (EmployeeRecord) in;
    erec.getEmprecord().setEmpSsn(BigDecimal.valueOf(99999));

    return (erec);
}

/*****
 * User exit for before business logic processing for a POST
 * request. This is called after the user activates a button
 * on the HTML form, but before the action associated with the
 * button is performed.
 */

public DataView doPostSetup (DataView in, HttpSession
session)
{
    EmployeeRecord erec;

    // Overrides the default behavior

    // validate the Social Security Number field
    erec = (EmployeeRecord) in;

    if (!isSsnValid(erec.getEmpRecord().getEmpSsn()))
    {
        // The SSN is not valid
        throw new Error ("Invalid Social Security Number:"
            = erec.getEmprecord().getEmpSsn());
    }

    return (erec);
}

/*****
 * User exit for after business logic processing for a POST
 * request. This is called after the action is performed for

```

```
* the button on the HTML form is activated by the user.
*/

public DataView doPostFinal(DataView in HttpSession session)
{
    // Overrides the default behavior

    // Nothing to do here

    return (in);
}

// Private functions

/*****
 * Validates an SSN field.
 *
 * @return
 * True if the SSN is valid, otherwise false.
 */

private boolean isSsnValid(final BigDecimal ssn)
{
    if (ssn.longValue() < 100000000)
    {
        // Oops, the SSN should not have a leading zero
        return (false);
    }
    else
        return (true);
}
}

//End ExtSampleServlet.java
```

Once the `ExtSampleServlet` class has been written, it and the other servlet Java source files must be compiled and deployed in the same manner as other servlets.

Generating a Client Enterprise Java Bean-Based Application

This type of application produces Java classes that comprise an EJB application. The class methods are invoked from requests originating from other EJB classes and transfer data records to and from the mainframe (remote system). From the viewpoint of the mainframe, the Java classes act as a remote DTP or IMS client. From the viewpoint of the EJB classes, they act as regular EJB classes.

Generating a Client Enterprise Java Bean-based application consists of the following steps:

1. Creating a script.
2. Processing the script.
3. Working with the generated files.
4. Customizing the application.
5. Compiling and deploying.

The following topics discuss these steps and offer examples of each.

Creating a Script

In order to produce an EJB-based application, the script file that defines your DataViews must be enhanced to describe both the mainframe services accessed and the EJB that will access them. A service description might look like the following listing example.

Listing 4-13 Sample service Description

```
service getSalary  
  
    accepts EmployeeRecord  
    returns EmployeeRecord
```

This sample listing defines a service named `getSalary` that accepts an input buffer of type `EmployeeRecord` and returns an output buffer of type `EmpDataView`. It is this service name which also requires an entry in the `jcrmgw.cfg` file.

An EJB that uses this service might be defined like the following listing.

Listing 4-14 Sample `getSalary` Service Definition

```
client ejb MyEJBName MyEJBHome
{
    method getPay is service getSalary
}
```

This listing defines a Java bean class named `MyEJBName` with a method named `getPay`. The method corresponds to service name `getSalary`. The EJB home will be registered in Java Naming and Directory Interface (JNDI) under the name `MyEJBHome`.

The following listing shows the contents of a complete script file for defining a client EJB application.

Listing 4-15 Sample Client EJB Script

```
1 #-----
2 # empclient.egen
3 #   JAM script for an employee record.
4 #
5 #   $Id: empclient.egen,v 1.1 2000/01/25  18:34:14 david Exp$
6 #-----
7
8 # DataViews (typed data records)
9
10 view sample.EmployeeRecord (Comment 1)
11     from emprec.cpy
12
13 # Services
14
15 service sampleCreate (Comment 2)
16     accepts EmployeeRecord
17     returns EmployeeRecord
18
19 service sampleRead (Comment 2)
```

```
20     accepts EmployeeRecord
21     returns EmployeeRecord
22
23 # Clients and servers
24
25 client ejb sample.SampleClient my.sampleBean (Comment 3)
26 {
27     method newEmployee (Comment 4)
28         is service sampleCreate
29
30     method readEmployee (Comment 4)
31         is service sampleRead
32 }
33
34 # End
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-6 Script Comments

Comment 1	Defines a DataView class, specifying its corresponding copybook source file and its package name.
Comment 2	Defines a service function and its input and output parameter types.
Comment 3	Defines a client EJB class and its home name.
Comment 4	Defines a client class method and its service name.

Processing the Script

Issuing the following command will process the script.

Listing 4-16 Sample Script Process Command

```
egencobol empclient.egen
emprec.cpy, Lines: 21, Errors: 0, Warnings: 0
Generating sample.EmployeeRecord...
Generating group emp-name
Generating group emp-addr
Generating SampleClient...
```

Working with Generated Files

This script command generates the following files.

Table 4-7 Sample Script Generated Files

File	Content
SampleClient.java	Source for the EJB remote interface.
SampleClientBean.java	Source for the EJB implementation.
SampleClientHome.java	Source for the EJB home interface.
EmployeeRecord.java	Source for the DataView object.
SampleClient-jar.xml	Sample deployment descriptor
wl-SampleClient-jar.xml	Sample WebLogic deployment information

SampleClient.java Source File

The following listing shows the contents of the generated `SampleClient.java` source file.

Listing 4-17 Sample SampleClient.java Contents

```
// SampleClient.java
//
// EJB Remote Interface generated by eGenCobol on 24-Jan-2000.

package sample;

// Imports

import javax.ejb.EJBObject;
...

/** Remote Interface for SampleClient EJB. */
public interface SampleClient (Comment 1)
    extends EJBObject
{
    // newEmployee (Comment 2)
    EmployeeRecord newEmployee (EmployeeRecord commarea)
        throws RemoteException, UnexpectedException;

    readEmployee (Comment 2)
    EmployeeRecord readEmployee (EmploymentRecord commarea)
        throws RemoteException, UnexpectedException;
}

// End SampleClient.java
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-8 Script Comments

Comment 1	Defines an EJB interface.
Comment 2	Methods to convert a raw COMMAREA into a Java DataView object.

SampleClientBean.java Source File

The following listing shows the contents of the generated SampleClientBean.java source file.

Listing 4-18 Sample SampleClientBean.java Contents

```
// SampleClientBean.java
//
// EJB generated by eGenCobol on 24-Jan-2000.

package sample;

//Imports

import com.bea.jam.egen.egenClientBean;
...

/** EJB implementation. */
public class SampleClientBean (Comment 1)
    extends egenClientBean
{
    // newEmployee

    public EmployeeRecord newEmployee (EmployeeRecord commarea)
        throws IOException, snaException (Comment 2)
    {
        ...
    }

    //readEmployee

    public EmployeeRecord readEmployee (EmployeeRecord commarea)
        throws IOException, snaException (Comment 2)
    {
        ...
    }
}

// End SampleClientBean.java
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-9 Script Comments

Comment 1	Defines an EJB client bean.
-----------	-----------------------------

Table 4-9 Script Comments

Comment 2	The methods convert a raw COMMAREA into a Java DataView object.
-----------	-----------------------------------------------------------------

SampleClientHome.java Source File

The following listing shows the contents of the generated SampleClientHome.java source file.

Listing 4-19 Sample SampleClientHome.java Contents

```
// SampleClientHome.java
//
// EJB Home interface generated by eGenCobol on 24-Jan-2000.

package sample;

// Imports

import javax.ejb.EJBHome;
...

/** Home interface for SampleClient EJB. */
public interface SampleClientHome (Comment 1)
    extends EJBHome
{
    // create

    SampleClient create()
        throws CreateException, RemoteException;
}

// End SampleClientHome.java
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-10 Script Comments

Comment 1	Defines an EJB home interface.
-----------	--------------------------------

SampleClient-jar.xml Source File

The following listing shows the contents of the SampleClient-jar.xml source file. To use this file, copy it to ejb.jar.xml.

Listing 4-20 Sample SampleClient-jar.xml Contents

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SampleClient</ejb-name>
      <home>sample.SampleClientHome</home>
      <remote>sample.SampleClient</remote>
      <ejb-class>sample.SampleClientBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>SampleClient</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

wl-SampleClient-jar.xml Source File

The following listing shows the contents of the wl-SampleClient-jar.xml source file. To use this file, copy it to weblogic-ejb-jar.xml.

Listing 4-21 Sample wl-SampleClient-jar.xml Contents

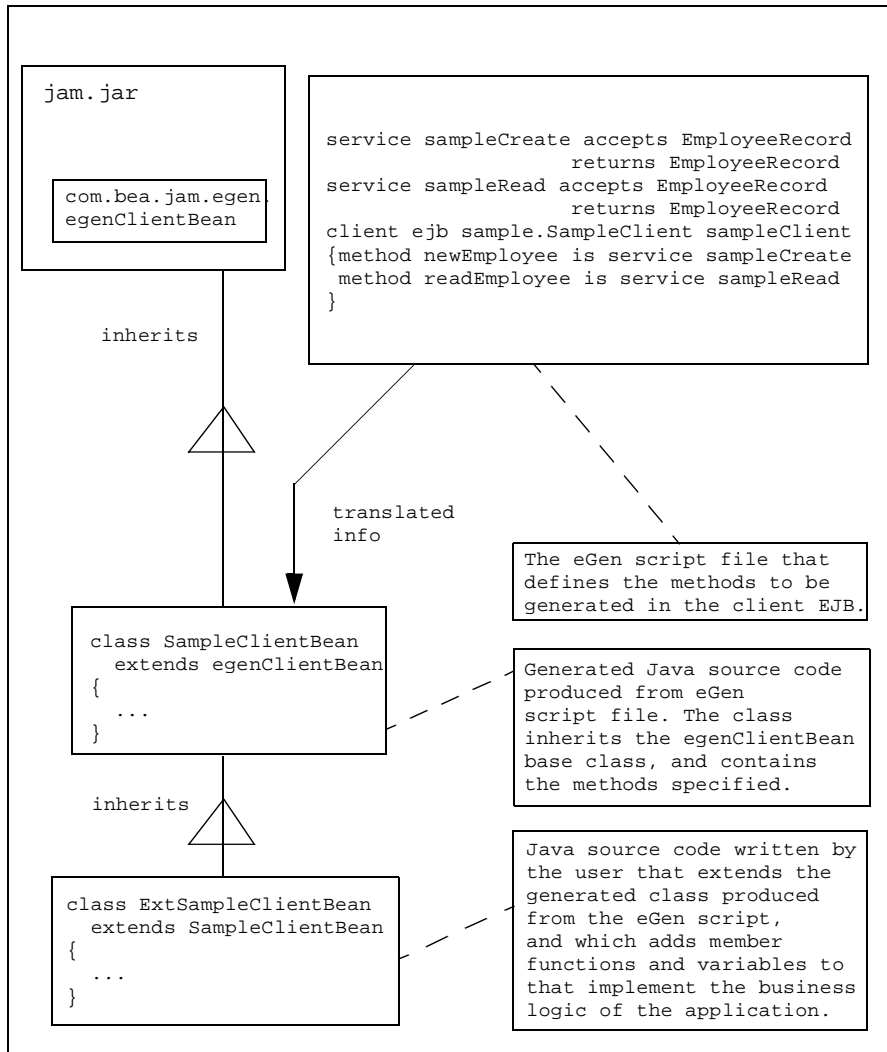
```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD
WebLogic 5.1.0 EJB//EN"
'http://www.bea.com/servers/wls510/dtd/weblogic-ejb-jar.dtd'>
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>SampleClient</ejb-name>
    <caching-descriptor>
      <max-beans-in-free-pool>50</max-beans-in-free-pool>
    </caching-descriptor>
    <jndi-name>my.sampleBean</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

Customizing an Enterprise Java Bean-Based Application

Unlike the servlet applications, the generated Java classes produced for EJB applications are intended for customization.

This section describes the way that generated client EJB code can be customized.

The following figure illustrates the relationships and inheritance hierarchy between the JAM classes comprising the application.

Figure 4-5 The JAM Client EJB Class Hierarchy

The generated Java code for a client EJB application is a class that inherits class `egenClientBean`. The `egenClientBean` class is provided in the JAM distribution jar file.

This base class, illustrated in the following listing, is provided in the jam.jar file and provides the basic framework for an EJB. It provides the required methods for a Stateless Session EJB.

Listing 4-22 egenClientBean.java **Base Class**

```
//=====
// egenClientBean.java
//   The base class for generated client EJB's.
//
//-----

package com.bea.jam.egen;

abstract public class egenClientBean
    implements SessionBean
{
    //Implementation of ejbActivate(), ejbRemove(),
    // ejbPassivate(), ejbCreate() and setSessionContext()
    ...

    /**
     * Call a service by name through the jcrmgw.
     *
     * @exception bea.sna.jcrmgw.snaException For Gateway errors
     * @exception java.io.IOException For data translation
     *         errors.
     */
    protected byte[] callService(String service, byte[] in)
        throws snaException, IOException
    {
        // Low level gateway access code
        ...
    }

    // Variables

    protected SessionContext m_context;
    protected transient Properties m_properties;
}

// End egenClientBean.java
```

The generated class, illustrated in the following Listing, adds the methods specific to this EJB.

Listing 4-23 Generated SampleClientBean.java Class

```

// SampleClientBean.java
//
// EJB generated by eGenCobol on Feb 2, 2000.
//

package Sample;

...

/**
 * EJB implementation.
 */
public class SampleClientBean extends egenClientBean
{
    // readEmployee
    //
    public EmployeeRecord readEmployee (EmployeeRecord commarea)
        throws IOException, snaException
    {
        // Make the remote call.
        //
        ...
    }

    // newEmployee
    //
    public EmployeeRecord newEmployee (EmployeeRecord commarea)
        throws IOException, snaException
    {
        // Make the remote call.
        //
        ...
    }
}

// END SampleClientBean.java

```

The following listing illustrates an example ExtSampleClientBean class that extends the generated SampleClientBean class, adding a validation function (isSsnValid()) for the emp-ssn (m_empSsn) field of the DataView EmployeeRecord class. The four methods are overridden by the methods in the extended class. If the emp-ssn field is determined to be invalid, an exception is thrown. Otherwise, the original function is called to perform the mainframe operation.

Listing 4-24 Example ExtSampleClientBean.java Class

```
//=====
// ExtSampleClientBean.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.SampleClientBean;

/*****
 * Extends the SampleClientBean EJB class, adding additional business logic.
 */

public class ExtSampleClientBean
    extends SampleClientBean
{
    // Public functions

    /*****
     * Read an employee record.
     */

    public EmployeeRecord readEmployee(EmployeeRecord commarea)
        throws RemoteException, UnexpectedException, IOException, snaException
    {
        EmployeeRecord  errec = (EmployeeRecord) commarea;

        if (!isSsnValid(errec.getEmpRecord().getEmpSsn()))
        {
            // The SSN is not valid.
            throw new Error("Invalid Social Security Number: "
                + errec.getEmpRecord().getEmpSsn());
        }

        // Make the remote call.
        return super.readEmployee(commarea);
    }
}
```

```

/*****
 * Create a new employee record.
 */

public EmployeeRecord newEmployee(EmployeeRecord commarea)
    throws IOException, snaException
{
    EmployeeRecord  erec = (EmployeeRecord) commarea;

    if (!isSsnValid(erec.getEmpRecord().getEmpSsn()))
    {
        // The SSN is not valid.
        throw new Error("Invalid Social Security Number:"
            + erec.getEmpRecord().getEmpSsn());
    }

    // Make the remote call.
    return super.newEmployee(commarea);
}

// Private Functions

/*****
 * Validate an SSN field.
 *
 * @return
 * True if the SSN is valid, otherwise false.
 */

private boolean isSsnValid(final BigDecimal ssn)
{
    if (ssn.longValue() < 100000000)
    {
        // Oops, appears to be less than 9 digits
        return false;
    }
    return true;
}

}

// End ExtSampleClientBean.java

```

When the ExtSampleClientBean class has been written, it and the other EJB Java source files must be compiled and deployed in the same manner as other EJBs. Prior to deploying, the deployment descriptor must be modified; *the ejb-class* property must be set to the name of your extended EJB implementation class.

Compiling and Deploying

Refer to the BEA WebLogic server documentation for more information. The sample file provided with the WebLogic Server contain build script for reference.

Generating a Server Enterprise Java Bean-Based Application

This type of application produces Java classes that comprise an EJB application, similar to a Client EJB application, but acting as a remote server from the viewpoint of the mainframe. The classes process service requests originating from the mainframe (remote) system, known as “inbound” requests, and transfer data records to and from the mainframe. From the viewpoint of the Java classes, they receive EJB method requests. From the viewpoint of the mainframe application, it invokes remote DPL or IMS programs.

Generating a Server Enterprise Java Bean-based application consists of the following steps:

1. Creating a script.
2. Processing the script.
3. Working with the generated files.
4. Customizing the application.
5. Compiling and deploying.

The following topics discuss these steps and offer examples of each.

Creating a Script

The following listing shows the contents of a complete script for defining a server EJB application.

Listing 4-25 Sample Server EJB Script

```
1 #-----
2 # empserver.egen
3 #   JAM script for an employee record.
4 #
5 # $Id: empserver.egen, v 1.1 2000/01/21 23:20:40
6 #-----
7
8 # DataViews (typed data records)
```

```
9
10  view sample.EmployeeRecord (Comment 1)
11      from emprec.cpy
12
13  # Clients and servers (Comment 2)
14
15  server ejb sample.SampleServer my.sampleServer (Comment 3)
16  {
17      method newEmployee (EmployeeRecord) (Comment 4)
18          returns EmployeeRecord
19  }
20
21  # End
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-11

Comment 1	Defines a DataView class, specifying its corresponding copybook source file and its package name.
Comment 2	Defines a server EJB class.
Comment 3	my.sampleServer is the home interface identifier for this bean. This value must be included in an entry in the local Services section of the jcrmgw.cfg file for the Java gateway.
Comment 4	Defines a server class method and its parameter.

Processing the Script

Issuing the following command will process the script.

Listing 4-26 Sample Script Process Command

```
egencobol empserver.egen
emprec.cpy, Lines: 21, Errors: 0, Warnings: 0
Generating sample.EmployeeRecord...
Generating group emp-name
```

```
Generating group emp-addr  
Generating SampleServer...
```

Generated Files

This script command generates the following files.

Table 4-12 Sample Script Generated Files

File	Content
SampleServer.java	Source for the EJB remote interface.
SampleServerBean.java	Source for the EJB implementation.
SampleServerHome.java	Source for the EJB home interface.
EmployeeRecord.java	Source for the DataView object.
SampleServer-jar.xml	Sample deployment descriptor
wl-SampleServer-jar.xml	Sample WebLogic deployment information

SampleServer.java Source File

The following listing shows the content of the generated SampleServer.java source file.

Listing 4-27 Sample SampleServer.java Contents

```
// SampleServer.java  
//  
// EJB Remote Interface generated by eGenCobol on 24-Jan-2000.  
  
package sample;  
  
// Imports  
  
import javax.ejb.EJBObject;  
...  
  
/** Remote Interface for SampleServer EJB. */
```

```
public interface SampleServer
    extends gwObject
{
    //dispatch
    byte[] dispatch(byte[] commarea, Object future)
        throws RemoteException, UnexpectedException;
}

// End SampleServer.java
```

SampleServerBean.java Source File

The following listing shows the contents of the generated SampleServerBean.java source file.

Listing 4-28 Sample SampleServerBean.java Contents

```
// SampleServerBean.java
//
EJB generated by eGenCobol on 24-Jan-2000.

package Sample;

// Imports

import com.bea.jam.egen.EgenServerBean;
...

/** EJB implementation. */

public class SampleServerBean
    extends EgenServerBean
{
    // dispatch
    public byte[] dispatch (byte[] commarea, Object future)
        throws IOException
    {
        ...
    }

    /**
     * Do the actual work for a newEmployee operation.
     * NOTE: This routine should be overridden to do actual work
     */
    EmployeeRecord newEmployee (EmployeeRecord commarea)
```

```
        {
            return new EmployeeRecord();
        }
    }
}

//End SampleServerBean.java
```

SampleServerHome.java Source File

The following listing shows the contents of the generated SampleServerHome.java source file.

Listing 4-29 Sample SampleServerHome.java Contents

```
// SampleServerHome.java
//
// EJB Home interface generated by eGenCobol on 24-Jan-2000.

package Sample;

//Imports

import javax.ejb.EJBHome;
...

/** Home interface for SampleServer EJB. */

public interface SampleServerHome
    extends EJBHome
{
    //create
    SampleServer create()
        throws CreateException, RemoteException;
}

// End SampleServerHome.java
```

SampleServer-jar.xml Source File

The following listing shows the contents of the generated SampleServer-jar.xml source file. To use this file, copy it to ejb.jar.xml

Listing 4-30 Sample SampleServer-jar.xml Contents

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SampleServer</ejb-name>
      <home>sample.SampleServerHome</home>
      <remote>sample.SampleServer</remote>
      <ejb-class>sample.SampleServerBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>SampleServer</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

wl-SampleServer-jar.xml Source File

The following listing shows the contents of the generated wl-SampleServer-jar.xml source file. To use this file, copy it to weblogic-ejb-jar.xml

Listing 4-31 Sample wl-SampleServer-jar.xml Contents

```
<?xml version="1.0"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD
WebLogic 5.1.0 EJB//EN"
'http://www.bea.com/servers/wls510/dtd/weblogic-ejb-jar.dtd'>
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>SampleServer</ejb-name>
```

```
< caching-descriptor>
  < max-beans-in-free-pool>50< /max-beans-in-free-pool>
< /caching-descriptor>
  < jndi-name>my.sampleBean< /jndi-name>
< /weblogic-enterprise-bean>
< /weblogic-ejb-jar>Script Comments
```

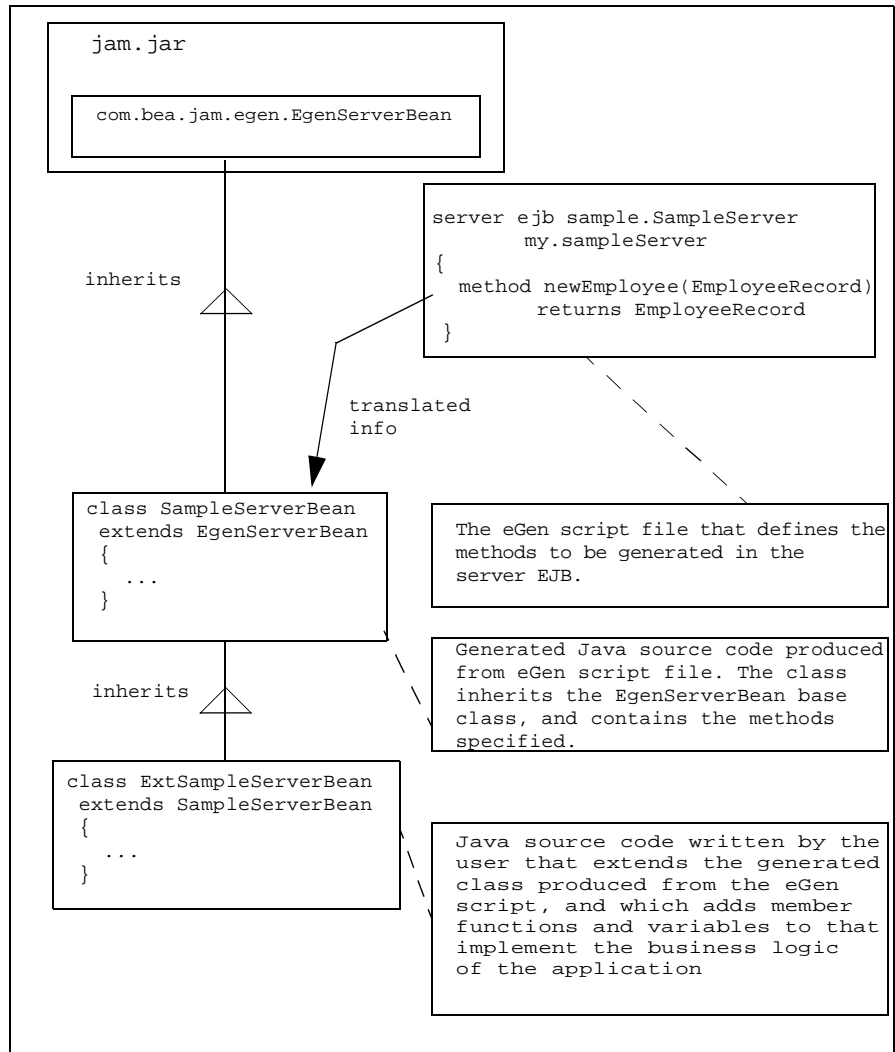
Customizing a Server Enterprise Java Bean-Based Application

The generated server enterprise Java bean-based applications are only intended for customizing, since they perform no real work without customization.

This section describes the way that generated server EJB code can be customized.

The following figure illustrates the relationships and inheritance hierarchy between the JAM classes comprising the application.

Figure 4-6 The JAM Server EJB Class Hierarchy



The generated Java code for a client EJB application is a class that inherits class `EgenServerBean`. The `EgenServerBean` class is provided in the JAM distribution jar file. This base class, illustrated in the following listing, provides the basic framework for an EJB. It provides the required methods for a Stateless Session EJB.

Listing 4-32 EgenServerBean.java Base Class

```
//=====
// EgenServerBean.java
//      The base class for generated server EJB's.
//
//=====

package com.bea.jam.egen;

abstract public class EgenServerBean
    implements SessionBean
{
    // Implementation of ejbActivate(), ejbRemove(),
    // ejbPassivate(),
    // setSessionContext() and ejbCreate().
    ...

    // Variables

    protected transient SessionContext m_context;
    protected transient Properties m_properties;
}

// End EgenServerBean.java
```

The generated class, illustrated in the following listing, adds the methods specific to this EJB.

Listing 4-33 Generated SampleServerBean.java Class

```
// SampleServerBean.java
//
// EJB generated by eGenCobol on 03-Feb-00.
//

package Sample;

//Imports
//
import java.io.IOException;
import java.util.Hashtable;
import com.bea.sna.jcrmgw.snaException;
import com.bea.base.io.MainframeWriter
```

```
import com.bea.base.io.MainframeReader;
import com.bea.jam.egen.EgenServerBean;
import com.bea.jam.egen.InboundDispatcher;

/**
 * EJB implementation
 */
public class SampleServerBean extends EgenServerBean
{
    // dispatch
    //
    public byte[] dispatch(byte[] commarea, Object future)
        throws IOException
    {
        EmployeeRecord    inputBuffer
                        = new EmployeeRecord (new
                                                MainframeReader (commarea));
        EmployeeRecord    result = newEmployee (inputBuffer);
        return result.toByteArray (new MainframeWriter());
    }

    /**
     * Do the actual work for a newEmployee operation.
     * NOTE: This routine should be overridden to do actual work
     */
    EmployeeRecord newEmployee(EmployeeRecord commarea)
    {
        return new EmployeeRecord();
    }
}

// END SampleServerBean.java
```

The following listing shows an example `ExtSampleServerBean` class that extends the generated `SampleServerBean` class, providing an implementation of the `newEmployee()` method. The example method prints a message indicating that a `newEmployee` request has been received.

Listing 4-34 Sample ExtSampleServerBean.java Contents

```
// ExtSampleServerBean.java
//

package sample;

/**
 * EJB implementation
 */
public class ExtSampleServerBean extends SampleServerBean
{
    public EmployeeRecord newEmployee (EmployeeRecord in)
    {
        System.out.println("New Employee: " +
            +in.getEmpRecord().in.getEmpName().getEmpNameFirst()
            + " "
            + in.getEmpRecord().getEmpname().getEmpNameLast());
        return in;
    }
}

// END ExtSampleServerBean.java
```

Once the ExtSampleServerBean class has been written, it and the other EJB Java source files must be compiled and deployed in the same manner as other EJBs. Before deploying, the deployment descriptor must be modified; the *ejb-class* must be set to the name of your extended EJB implementation class.

Compiling and Deploying

Refer to the BEA WebLogic server documentation for more information. The sample file provided with the WebLogic Server contain build script for reference.

Generating a Stand-Alone Client Application

This type of application produces simple Java classes that perform the appropriate conversions of data records sent between Java and the mainframe, but without all of the EJB support methods. These classes are intended to be lower-level components upon which more complicated applications are built.

Generating a stand-alone application consists of the following steps:

1. Generating a script.
2. Processing the script.
3. Working with the generated files.
4. Customizing the application.

The following listing shows the contents of a complete script for defining a stand-alone client class with multiple services.

Listing 4-35 Sample Stand-Alone Client Class Script

```
1  #-----
2  # empclass.egen
3  #   JAM script for an employee record.
4  #
5  # $Id: empclass.egen, v 1.1 2000/01/21 23:20:40
6  #-----
7
8  # DataViews (typed data records)
9
10 view sample.EmployeeRecord (Comment 1)
11     from emprec.cpy
12
13     # Services
14
15     service sampleCreate (Comment 2)
16         accepts EmployeeRecord
17         returns EmployeeRecord
18
19     service sampleRead (Comment 2)
20         accepts EmployeeRecord
21         returns EmployeeRecord
22
```

```

23  service sampleUpdate (Comment 2)
24      accepts EmployeeRecord
25      returns EmployeeRecord
26
27  service sampleDelete (Comment 2)
28      accepts EmployeeRecord
29      returns EmployeeRecord
30
31  # Clients and servers
32
33  client class sample.SampleClass (Comment 3)
34  {
35      method newEmployee (Comment 4)
36          is service sampleCreate
37
38      method readEmployee (Comment 4)
39          is service sampleRead
40  }
41
42  # End

```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-13 Script Comments

Comment 1	Defines a DataView class, specifying its corresponding copybook source file and its package name.
Comment 2	Defines a service function and its input and output parameter types.
Comment 3	Defines a simple client class.
Comment 4	Defines a client class method and its parameter types.

Processing a Script

Issuing the following command will process the script.

Listing 4-36 Sample Script Process Command

```
egencobol empclass.egen
emprec.cpy, Lines: 21, Errors: 0, warnings: 0
Generating sample.EmployeeRecord...
Generating group emp-name
Generating group emp-addr
Generating SampleClass...
```

Generated Files

This script command generates the following files.

Table 4-14 Sample Script Generated Files

File	Content
SampleClass.java	Source for the sample class.
EmployeeRecord.java	Source for the DataView class.

SampleClass.java Source File

The following listing contains the generated SampleClass.java source file.

Listing 4-37 Sample SampleClass.java Source File

```
// SampleClass.java
//
// Client class generated by eGenCobol on 24-Jan-2000.

package sample;(Comment 1)

// Imports

import com.bea.jam.egen.EgenClient;
...

/* Mainframe client class. */

public class SampleClass (Comment 2)
    extends EgenClient
```

```
{
    // newEmployee
    public EmployeeRecord newEmployee (EmployeeRecord commarea)
        throws IOException, snaException (Comment 3)
    {
        ...
    }

    // readEmployee
    public EmployeeRecord readEmployee (EmployeeRecord commarea)
        throws IOException, snaException (Comment 3)
    {
        ...
    }
}

// End SampleClass.java
```

Script Comments

The following numbered comments refer to the numbered comments in the prior listing.

Table 4-15 Script Comments

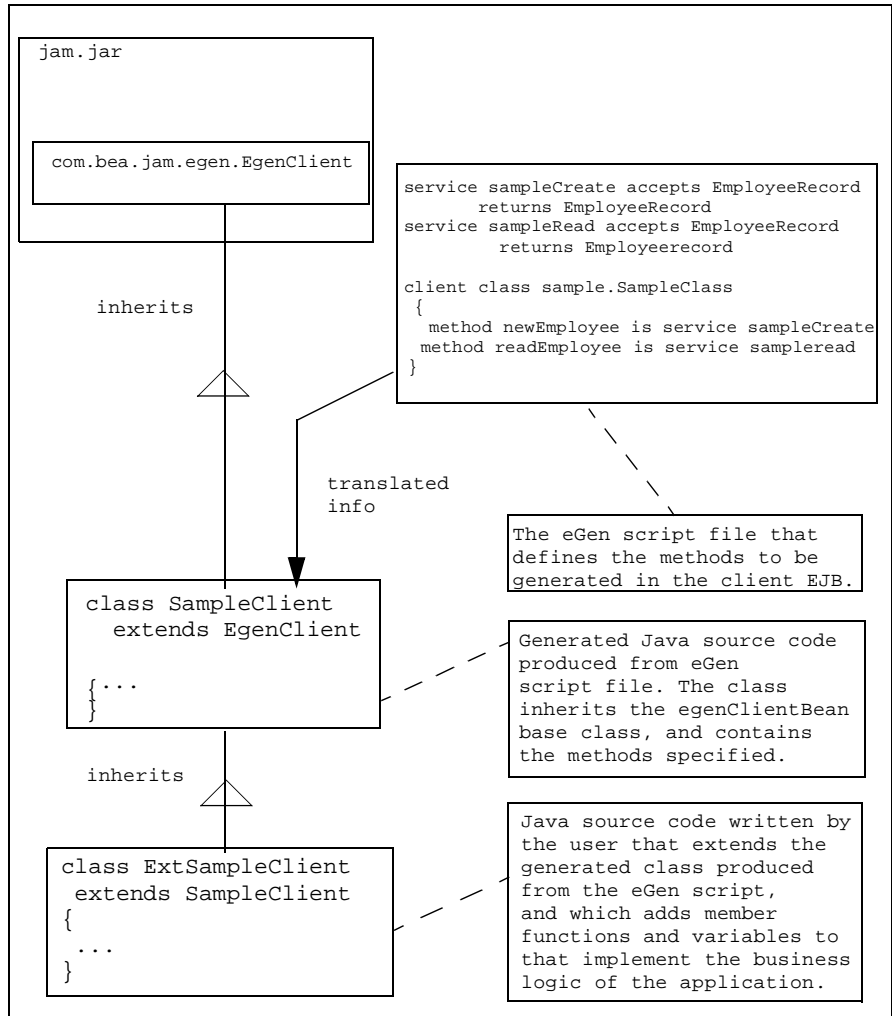
Comment 1	The package name is defined in the eGen script.
Comment 2	The data record is encapsulated in a class that extends the EgenClient class.
Comment 3	The methods convert a raw COMMAREA into a Java DataView object.

Customizing a Stand-Alone Java Application

The stand-alone client class model is the simplest JAM code generation model both in terms of the code generated and customizing the generated code.

The following figure illustrates the relationships and inheritance hierarchy between the JAM classes comprising the application.

Figure 4-7 The JAM Client EJB Class Hierarchy



The generated Java code for a client class application is a class that inherits class `EgenClient`. The `EgenClient` class is provided in the JAM distribution jar file. This base class, illustrated in the following listing provides the basic framework for a client to the `jcrmgw`. It provides the required methods for accessing the gateway.

Listing 4-38 Generated EgenClient.java Class

```
//=====
// EgenClient.java
//      Basic functionality for clients of the jcrmgw
//
//-----

package com.bea.jam.egen;

public class EgenClient
{
    public byte[] callService(String service, byte[] in)
        throws snaException, IOException
    {
        // make a mainframe request through the gateway.
        ...
    }
}

// End egenClientBean.java
```

The generated class, illustrated in the following listing, adds the methods specific to the users application

Listing 4-39 Sample SampleClient.java Class

```
// SampleClass.java
//
// Client class generated by eGenCobol on 02-Feb-00.
//

package sample;

// Imports
//
import java.io.IOException;
import com.bea.jam.egen.EgenClient;
import com.bea.sna.jcrmgw.snaException;
import com.bea.base.io.MainframeWriter;
import com.bea.base.io.MainframeReader;
```

```
/**
 * Mainframe client class.
 */
public class SampleClass extends EgenClient
{
    // newEmployee
    //
    public EmployeeRecord newEmployee(EmployeeRecord commarea)
        throws IOException, SnaException
    {
        // Make the remote call.
        //
        byte[] inputBuffer = commarea.toByteArray(new
            MainframeWriter());
        byte[] rawResult = callService("sampleCreate",
            inputBuffer);
        EmployeeRecord result =
            new EmployeeRecord(new
                MainframeReader(rawResult));
        return result;
    }

    // readEmployee
    //
    public EmployeeRecord readEmployee(EmployeeRecord commarea)
        throws IOException, SnaException
    {
        // Make the remote call.
        //
        byte[] inputBuffer = commarea.toButeArray(new
            MainframeWriter());
        byte[] rawResult = callService("sampleRead", inputBuffer);
        EmployeeRecord result =
            new EmployeeRecord(new MainframeReader(rawResult));
        return result
    }
}

// End SampleClass.java
```

Your class, which extends or uses the `SampleClient` class, simply overrides or calls these methods to provide additional business logic, modifying the contents of the `DataView`. It may also add additional methods, if desired.

The following listing shows an example `ExtSampleClass` class that extends the generated `SampleClient` class.

Listing 4-40 Sample ExtSampleClient.java Contents

```

// ExtSampleClient.java
//

package sample;

// Imports
//
import java.io.IOException;
import com.bea.jam.egen.egenClientBean;
import com.bea.sna.jcrmgw.snaException;
import com.bea.base.io.MainframeWriter;
import com.bea.base.io.MainframeReader;

/**
 * Extended Sample Class
 */
public class ExtSampleClient extends SampleClass
{
    // deleteEmployee
    //
    public EmployeeRecord deleteEmployee(EmployeeRecord
        commarea)
        throws IOException, snaException
    {
        EmployeeRecord ereco=(EmployeeRecord) in;
        if (!isSsnValid(ereco.getEmpRecord().getEmpSsn()))
        {
            // The SSN is not valid.
            throw new Error("Invalid Social Security Number:"+
                ereco.getEmpRecord().getEmpSsn());
        }

        // Make the remote call.
        //
        return super.deleteEmployee(commarea);
    }

    //updateEmployee
    //
    public EmployeeRecord updateEmployee(EmployeeRecord
        commarea)
        throws IOException, snaException
    {
        EmployeeRecord ereco = (EmployeeRecord) in;
        if (!isSsnValid(ereco.getEmpRecord().getEmpSsn()))
        {
            The SSN is not valid.

```

```
        throw new Error ("Invalid Social Security Number:" +
            erec.getEmpRecord().getEmpSsn());
    }

    // Make the remote call.
    //
    return super.updateEmployee(commarea);
}

// readEmployee
//
public EmployeeRecord readEmployee(EmployeeRecord commarea)
    throws IOException, snaException
{
    EmployeeRecord erec = (EmployeeRecord)in;
    if (!isSsnValid(erec.getEmpRecord().getEmpSsn()))
    {
        // The SSN is not valid.
        throw new Error("Invalid Social Security
            Number:" +
                erec.getEmpRecord().getEmpSsn());
    }

    // Make the remote call.
    //
    return super.readEmployee(commarea);
}

//newEmployee
//
public EmployeeRecord newEmployee(EmployeeRecord commarea)
    throws IOException, snaException
{
    EmployeeRecord erec = (EmployeeRecord) in;
    if (!isSsnValid(erec.getEmpRecord().getEmpSsn()))
    {
        // The SSN is not valid.
        throw new Error("Invalid Social Security Number:" +
            erec.getEmpRecord().getEmpSsn());
    }

    // Make the remote call.
    //
    return super.newEmployee(commarea);
}

// Private functions
```

```

/*****
 * Validates an SSN field.
 */

private boolean isSsnValid(BigDecimal ssn)
{
    if (ssn.longValue() < 100000000)
    {
        // Ops, should not have a leading zero.
        return false;
    }

    return (true);
}

]

// END ExtSampleClient.java

```

Once the `ExtSampleClient` class has been written, it and the other Java source files must be compiled and placed on to your `WEBLOGICCLASSPATH`.

Using Client Diagnostic Features

JAM includes several features to support diagnosing problems with eGen-based client programs. While these facilities are not designed for use in a production environment, they should be useful during development. These features are enabled by adding the following settings to your `weblogic.properties` file.

<code>java.system.property.bea.jam.client.trace.enable</code>	Set to "true" to enable tracing of client requests.
<code>java.system.property.bea.jam.client.trace.codepage</code>	Set to the name of a codepage to be used for the character portion of the trace dump.

java.system.property.bea.jam.client.loopback	Set to "true" to bypass the gateway & simply loop the request bytes back to the client.
java.system.property.bea.jam.client.stub	Set to the full name of a class to be used as a gateway stub.

Client Traffic Tracing

When this feature is enabled, all requests from eGen clients are dumped to the WebLogic console in hexadecimal and characters. The following listing shows an example of an eGen client dump.

Listing 4-41 Dump of eGen Client Requests

```
----- Service: demoRead    Input data -----
00 00 00 00 0f e2 d4 c9 e3 c8 40 40 40 40 40 40    .....SMITH
40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
40 40 40 40 40 40 40 40 40 40 40 40 40 00 00 00    .
01 00 00 00 00 00 0f 00 00 00 00 0f 00 00 00 00    .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....
----- Service: demoRead    Output data -----
00 00 00 00 0f e2 d4 c9 e3 c8 40 40 40 40 40 40    .....SMITH
40 40 40 40 a7 40 40 40 40 40 40 40 40 40 40 40           x
40 40 40 a7 94 81 89 95 40 40 40 40 40 40 40 40           xmain
40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
40 40 40 40 40 40 40 40 40 40 40 40 40 40 00 00    ..
01 00 00 00 00 0f 00 00 00 00 0f                        .....
-----
```

Note that the dumps occur while the data is in mainframe format, and characters will usually be in some variety of EBCDIC. By default, the character data is converted using cp037, but that may be changed using another property setting.

Client Loopback

If this feature is enabled, all requests receive a response that is exactly equal to the request data. Note that this is done while the data is in mainframe format. If a service accepts one `DataView` subclass and returns a different one, this may result in a conversion failure trying to construct the result `DataView` subclass.

When this feature is enabled, no gateway is required and the `gwboot` startup may be removed from your `weblogic.properties` file.

Client Stub Operation

This feature enables you to replace the gateway with your own class, in effect providing a replacement for the entire target mainframe. This feature is valuable for testing or proof-of-concept situations where the mainframe connection is not available.

Your stub class must:

- Provide a constructor that takes no arguments.
- Be available on your `WEBLOGICCLASSPATH`.
- Contain a method for each service that is to be supported. This method must take some `DataView` subclass as its only argument and return a `DataView` subclass.

The client tracing feature can be used to help debug your stub class.

5 Deploying Applications

Deployment is the process of taking previously developed servlets and/or EJBs and installing them into a specific operational environment. In this case, the operational environment is your WebLogic system. This section describes how to deploy your own applications using the BEA WebLogic Java Adapter for Mainframe (JAM) software.

This section discusses the following topics:

- [Deploying Servlets](#)
- [Deploying Enterprise Java Beans](#)

Deploying Servlets

Please refer to the BEA WebLogic Server documentation for detailed instructions on deploying servlets.

The following steps provide an example of deploying a servlet by modifying the `weblogics.properties` file.

Perform the following steps to deploy a servlet:

1. Place a class file for the servlet and all other classes that it depends on into the servlet classpath. Be sure to include the class files for the DataView's that are used by the servlet.

2. Add a registration line to your `weblogics.properties` file. This line will specify both the URL that will be used to access the servlet, and the full package and name of the servlet class. For example, assuming your server is named `www.webstore.com`, the properties line `weblogic.httpd.register.widgets=webstore.WidgetServlet` will cause the class `webstore.WidgetServlet` to serve requests to the URL `http://www.webstore.com/widgets`.

```
weblogic.httpd.register.urlname=mypackage.MyServlet
```

This registration line causes the WebLogic Server to use the servlet class “`mypackage.MyServlet`” to serve all HTTP requests for a page at the URL name (on your host).

3. Add any corresponding remote service entries into the `jcrmgw.cfg` file. For example, if your servlet invokes a service named `widgetQuote`, then you would need something like the following in your configuration:

```
*JC_REMOTE_SERVICES
widgetQuote RDOM="myMainframe"
            RNAME="WIDQUOTE"
```

4. Restart the WebLogic Server.

There are other servlet deployment options, including hot-deploying a servlet into a running WebLogic server. Please refer to the WebLogic Server documentation for more details.

Deploying Enterprise Java Beans

Please refer to the BEA WebLogic Server documentation for detailed instructions on deploying EJBs. The WebLogic samples provide examples of build scripts for UNIZ and NT platforms.

Perform the following steps to deploy an EJB:

1. Create a jar file containing the class files for the EJB and any other classes needed by your EJB. This jar file must be constructed following the normal Java requirements for class jar files.

2. Rename the deployment XML files to the standard names (ejb-jar.xml and weblogic-ejb-jar.xml) and place them into the jar file. They must be placed into a subdirectory within the jar file named 'META-INF'
3. Use the weblogic ejb compiler (weblogic.ejbc) to build the container classes for you EJB. The command will look something like:

```
java weblogic.ejbc input.jar output.jar
```

4. Edit your weblogic.properties file to add the output jar file to the EJB deployment list:

```
weblogic.ejb.deploy=c:\my\path\output.jar
```

5. Add any corresponding local or remote service entries into the jcrmgw.cfg file. For example, if your EJB invokes a service named "widgetQuote", then you would need something like the following in your configuration:

```
*JC_REMOTE_SERVICES
widgetQuote  RDOM="myMainframe"
              RNAME="WIDQUOTE"
```

6. Restart BEA WebLogic Server.

EJBs may also be placed into jar files before being deployed, and may be hot-deployed into a running WebLogic Server. Please refer to the Weblogic Server documentation for more details.

If you wish to deploy more than one EJB in the same jar file, the deployment XML files must be combined. Refer to the WebLogic Server documentation.

Give special attention to classpath setting if you are also deploying a servlet that accesses this EJB or the same DataView subclass. You may need to place all common classes on your WEBLOGICCLASSPATH rather than on your servlet classpath.

6 Security

BEA WebLogic Java Adapter for Mainframe (JAM) supports the basic Application Program-to-Program Communication (APPC) style of sign-on security. You can configure a gateway to use one of three types of sign-on security for each link that is defined (refer to the [“JC_SNALINKS Section”](#) in [“Configuring the BEA WebLogic Java Adapter for Mainframe.”](#)). The selected level of security determines what combination of userid and password is used for transactions across the link.

JAM supports the following security options:

- **LOCAL**
All security is handled by the local system and the link itself has no security requirement.
- **IDENTIFY**
A userid is passed to the mainframe. This userid can originate with the client application or it can be a default userid supplied at Java gateway startup by the `-u` option in the WebLogic Server `properties` file as part of the `startupArgs` entry for the gateway.
- **VERIFY**
A userid and password are passed to the mainframe. The userid can originate with the client application or it can be a default userid supplied at Java gateway startup by the `-u` option in the WebLogic Server `properties` file as part of the `startupArgs` entry for the gateway. The password must be supplied by the client application.

In addition, an alternate mirror transaction is supported on each Distributed Program Link (DPL), which can be used to associate different Resource Access Control Facility (RACF) profiles with different services.

Refer to IBM RACF documentation for more specific information about establishing and administrating mainframe security.

7 Programming Reference

This section provides the rules that enable you to identify what form a generated Java class takes from a given COBOL copybook processed by the eGen COBOL tool. This facilitates your being able to correctly code any custom programs that make use of the generated classes.

The eGen COBOL tool maps a COBOL copybook into a Java class. The COBOL copybook contains a data record description. The eGen COBOL tool derives the generated Java class from the `com.bea.dmd.dataview.DataView` class (later referred to as `DataView`), which is provided on your BEA WebLogic Java Adapter for Mainframe (JAM) product CD-ROM in the `jam.jar` file.

This section gives you data mapping rules. The following topics are discussed:

- [Field Name Mapping Rules](#)
- [Field Type Mappings](#)
- [Group Field Accessors](#)
- [Elementary Field Accessors](#)
- [Array Field Accessors](#)
- [Fields with REDEFINES Clauses](#)
- [COBOL Data Types](#)
- [Other Access Methods for Generated DataView Classes](#)

You should find the COBOL terms in this section easy to understand; however, you may need to use a COBOL reference book or discuss the terms with a COBOL programmer. Also, you can process a copybook with the eGen COBOL tool and examine the produced Java code in order to understand the mapping.

Field Name Mapping Rules

When you process a COBOL copybook containing field names, they are mapped to Java names. This is performed by the eGen COBOL tool according to the following rules:

1. All alphabetic characters are mapped to lower case.
2. All dashes are removed and the character following the dash is mapped to upper case.
3. When a prefix is added to the name (as when creating a field accessor function name) the first character of the base name is mapped to upper case.

[Table 7-1](#) lists some mapping examples.

Table 7-1 Example Field Name Mapping from COBOL to Java and Accessor

COBOL Field Name	Java Base Name	Sample Accessor Name
EMP-REC	empRec	setEmpRec
500-REC-CNT	500RecCnt	set500RecCnt

Field Type Mappings

When you process a COBOL copybook with field types, the field types are mapped to Java field types. This is performed by the eGen COBOL tool according to the following rules:

1. Group fields map to `DataView` subclasses.
2. All alphanumeric fields are mapped to type `String`.
3. All edited numeric fields are mapped to type `String`.
4. All `SIGN IS LEADING`, `SIGN IS TRAILING`, `BLANK WHEN ZERO` or `JUSTIFIED RIGHT` fields are mapped to type `String`.
5. The types `COMP-1`, `COMP-2`, `COMP-5`, `COMP-X`, and `PROCEDURE-POINTER` fields are not supported (an error message is generated).
6. All `INDEX` fields are mapped to Java type `int`.
7. `POINTER` maps to Java type `int`.
8. All numeric fields with any digits to the right of the decimal point are mapped to type `BigDecimal`.
9. All `COMP-3` (packed) fields are mapped to type `BigDecimal`.
10. All other numeric fields are mapped as shown in [Table 7-2](#).

Table 7-2 Numeric Field Mapping

Number of Digits	Java Type
≤ 4	<code>short</code>
> 4 and ≤ 9	<code>int</code>
> 9 and ≤ 18	<code>long</code>
> 18	<code>BigDecimal</code>

Group Field Accessors

Each nested group field in a COBOL copybook is mapped to a corresponding `DataView` subclass. The generated subclasses are nested exactly as the COBOL groups in the copybook. In addition, the eGen COBOL tool generates a private instance variable of this class type and a `get` accessor.

For example, the following copybook:

```
10 MY-RECORD.  
    20 MY-GRP.  
        30 ALNUM-FIELD                PIC X(20).
```

Produces code similar to the following:

```
public MyGrp2V getMyGrp();  
public static class MyGrp2V extends DataView  
{  
    // Class definition  
}
```

Elementary Field Accessors

Each elementary field is mapped to a private instance variable within the generated DataView subclass. Access to this variable is accomplished by two accessors that are generated (set and get).

These accessors have the following forms:

```
public void setFieldName(FieldType value);  
public FieldType getFieldName();
```

Where:

FieldType
is described in the [“Field Type Mappings”](#) section.

FieldName
is described in the [“Field Name Mapping Rules”](#) section.

For example, the following copybook:

```
10 MY-RECORD.  
    20 NUMERIC-FIELD                PIC S9(5).  
    20 ALNUM-FIELD                PIC X(20).
```

Produces the accessors:

```
public void setNumericField(int value);  
public int getNumericField();
```

```
public void setAlnumField(String value);  
public String getAlnumField();
```

Array Field Accessors

Array fields are handled according to the other rules in this section, with the addition that each accessor takes an additional `int` argument that specifies which array entry is to be accessed, for example:

```
public void          setFieldName(int index, FieldType value);  
public FieldType    getFieldName(int index);
```

Array fields specified with the `DEPENDING ON` clause are handled the same as fixed-size arrays with the following special rules:

1. The accessors may be used to `get` or `set` any instance up to the maximum array index.
2. The controlling (`DEPENDING ON`) variable is evaluated when the `DataView` is converted to or from an external format, such as a mainframe format. The eGen COBOL tool converts only the array elements with subscripts less than the controlling value.

Fields with REDEFINES Clauses

Fields that participate in a `REDEFINES` set are handled as a unit. A private `byte[]` variable is declared to hold the underlying mainframe data, as well as a private `DataView` variable. Each of the redefined fields has an accessor or accessors. These accessors take more CPU overhead than the normal accessors because they perform conversions to and from the underlying `byte[]` data.

For example the copybook:

```
10 MY-RECORD.  
   20 INPUT-DATA.  
       30 INPUT-A                               PIC X(4).
```

```
30 INPUT-B PIC X(4).  
20 OUTPUT-DATA REDEFINES INPUT-DATA PIC X(8).
```

Produces Java code similar to the following:

```
private byte[] m_redef23;  
private DataView m_redef23DV;  
public InputDataV getInputData();  
public String getOutputData();  
public void setOutputData(String value);  
public static class InputDataV extends DataView  
{  
    // Class definition.  
}
```

COBOL Data Types

This section summarizes the COBOL data types supported by JAM software. [Table 7-3](#) lists the COBOL data item definitions recognized by the eGen COBOL tool. [Table 7-4](#) lists the syntactical features and data types recognized by the eGen COBOL tool. If a COBOL feature is unsupported, an error message is generated, unless it is listed as ignored in a table.

Table 7-3 Major COBOL Features

COBOL Feature	Support
IDENTIFICATION DIVISION	unsupported
ENVIRONMENT DIVISION	unsupported
DATA DIVISION	partially supported
WORKING-STORAGE SECTION	partially supported
<i>Data record definition</i>	supported
PROCEDURE DIVISION	unsupported
COPY	unsupported
COPY REPLACING	unsupported

Table 7-3 Major COBOL Features

COBOL Feature	Support
EJECT, SKIP1, SKIP2, SKIP3	supported

Table 7-4 COBOL Data Types

COBOL Type	Java Type
COMP, COMP-4, BINARY (<i>integer</i>)	short/int/long
COMP, COMP-4, BINARY (<i>fixed</i>)	BigDecimal
COMP-3, PACKED-DECIMAL	BigDecimal
COMP-5	not supported
COMP-X	not supported
DISPLAY <i>numeric (zoned)</i>	BigDecimal
BLANK WHEN ZERO (<i>zoned</i>)	String
SIGN IS LEADING (<i>zoned</i>)	String
SIGN IS LEADING SEPARATE (<i>zoned</i>)	String
SIGN IS TRAILING (<i>zoned</i>)	String
SIGN IS TRAILING SEPARATE (<i>zoned</i>)	String
<i>edited numeric</i>	String
COMP-1, COMP-2 (<i>float</i>)	not supported
<i>edited float numeric</i>	String
DISPLAY (<i>alphanumeric</i>)	String
<i>edited alphanumeric</i>	String
INDEX	int
POINTER	int

Table 7-4 COBOL Data Types

COBOL Type	Java Type
PROCEDURE-POINTER	not supported
JUSTIFIED RIGHT	not supported (ignored)
SYNCHRONIZED	not supported (ignored)
REDEFINES	supported
66 RENAMES	not supported
66 RENAMES THRU	not supported
77 level	supported
88 level (<i>condition</i>)	not supported (ignored)
<i>group record</i>	inner class
OCCURS (<i>fixed array</i>)	array
OCCURS DEPENDING (<i>variable-length array</i>)	array
OCCURS INDEXED BY	not supported (ignored)
OCCURS KEY IS	not supported (ignored)

Other Access Methods for Generated DataView Classes

JAM allows you to access DataView classes through several methods as described in the following sections:

- [Mainframe Access to DataView Classes](#)
- [XML Access to DataView Classes](#)
- [Hashtable Access to DataView Classes](#)

Mainframe Access to DataView Classes

This section describes how mainframe format data may be moved into and out of DataView classes. The eGen COBOL tool writes this code for you, so this information is provided as reference.

Mainframe format data may be extracted from a DataView class through the use of the MainframeWriter class. [Listing 7-1](#) shows a sample of code that may be used to perform the extraction.

Listing 7-1 Sample Code for Extracting Mainframe Format Data from a DataView Class

```
import com.bea.base.io.MainframeWriter;
import com.bea.dmd.dataview.DataView;

...

/**
 * Get mainframe format data from a DataView into a byte[].
 */
byte[] getMainframeData(DataView dv)
{
    try
    {
        MainframeWriter mw = new MainframeWriter();
        // To override the DataView's codepage, change the
        // above constructor call to something like:
        // ...new MainframeWriter("cp1234");

        return dv.toByteArray(mw);
    }
    catch (java.io.IOException e)
    {
        // Some conversion failure occurred...
    }
}
```

If you wish to override the codepage provided when the DataView was generated, you may provide another codepage as a String argument to the MainframeWriter constructor, as shown in the comment in [Listing 7-1](#).

Loading mainframe data into a `DataView` is a similar process, in this case requiring the use of the `MainframeReader` class. [Listing 7-2](#) shows a sample of code that may be used to perform the load.

Listing 7-2 Sample Code for Loading Mainframe Data into a `DataView` Class

```
import com.bea.base.io.MainframeReader;
import com.bea.dmd.dataview.DataView;

...

/**
 * Put a byte[] containing mainframe format data into a DataView.
 */
MyDataView putMainframeData(byte[] buffer)
{
    MainframeReader mr = new MainframeReader(buffer);
    // To override the DataView's codepage, change the above
    // constructor call to something like:
    // ...new MainframeReader("cpl234", buffer);

    MyDataView dv;

    try
    {
        // Construct a new DataView with the mainframe data.
        dv = new MyDataView(mr);

        // Or, to load a pre-existing DataView with mainframe
data.        // dv.mainframeLoad(mr);
    }
    catch (java.io.IOException e)
    {
        // Some conversion failure occurred.
    }

    return dv;
}
```

XML Access to DataView Classes

Facilities are provided to move XML data into and out of DataView classes. These operations are performed through the use of the `XmlLoader` and `XmlUnloader` classes.

- `XmlLoader` is used to load XML data into a `DataView`, `XmlUnloader` is used to unload data from a `DataView` into XML.
- If the eGen COBOL script used to produce the `DataView` specifies the "support xml" option, then both a DTD and an XML/Schema that describe the XML format for this `DataView` will also be produced.

[Listing 7-3](#) shows an example of the code to load XML data into a `DataView`.

Listing 7-3 Sample Code for Loading XML Data into a DataView

```
import com.bea.dmd.dataview.DataView;
import com.bea.dmd.dataview.XmlLoader;

...

void loadXmlData(String xml, DataView dv)
{
    XmlLoader xl = new XmlLoader();
    try
    {
        // Load the xml. Note that the xml argument may be either
        // a String or a org.w3c.dom.Element object.
        xl.load(xml, dv);
    }
    catch (Exception e)
    {
        // Some conversion error occurred.
    }
}
```

[Listing 7-4](#) shows an example of the code to unload a `DataView` into XML.

Listing 7-4 Sample Code for Unloading a DataView into XML

```
import com.bea.dmd.dataview.DataView;
import com.bea.dmd.dataview.XmlUnloader;

...

String unloadXmlData(DataView dv)
{
    XmlUnloader xu = new XmlUnloader();

    try
    {
        String xml = xu.unload(dv);
        return xml;
    }
    catch (Exception e)
    {
        // Some conversion error occurred.
    }
}
```

Hashtable Access to DataView Classes

JAM also provides facilities to load and unload DataView objects using Hashtable objects. Hashtable objects will most often be used to move data from one DataView to another similar DataView.

When DataView fields are moved into Hashtables, each field is given a key that is a string that reflects the location of the field within the original copybook data structure. [Listing 7-5](#) shows a sample of a COBOL Copybook.

Listing 7-5 Sample `emprec.cpy` COBOL Copybook

```
1      *-----
2      *  emprec.cpy
3      *      An employee record.
4      *-----
5
6      02      emp-record. (Comment 1)
```

```

7
8          04      emp-ssn                pic 9(9)  comp-3.
9
10         04      emp-name.
11             06      emp-name-last      pic x(15). (Comment 2)
12             06      emp-name-first     pic x(15).
13             06      emp-name-mi        pic x.
14
15         04      emp-addr. (Comment 3)
16             06      emp-addr-street     pic x(30).
17             06      emp-addr-st         pic x(2).
18             06      emp-addr-zip        pic x(9).
19
20      * End

```

The fields for the COBOL Copybook in [Listing 7-5](#) are stored into a Hashtable as shown in the following table.

Key String	Content Type
empRecord.empSsn	BigDecimal
empRecord.empName.empNameLast	String
empRecord.empName.empNameFirst	String
empRecord.empName.empNameMi	String
empRecord.empAddr.empAddrStreet	String
empRecord.empAddr.empAddrSt	String
empRecord.empAddr.empAddrZip	String

Code for Unloading and Loading Hashtables

Following is an example of the code used to **unload** a DataView into a Hashtable.

```
Hashtable ht = new HashtableUnloader().unload(dv);
```

Following is an example of the code used to **load** a Hashtable into an existing DataView.

```
new HashtableLoader().load(dv);
```

Rules for Unloading and Loading Hashtables

The basic rules of Hashtable **unloading** are:

- All data elements in the DataView will be placed into the Hashtable.
- Each data item will be stored as an object of its Java type. Elements of in/short/long type will be converted to Integer/Short/Long.
- Arrays will be mentioned at the appropriate level in the key as an index enclosed in "[", "]" pairs. For instance, if empAddr was an array then one key into the Hashtable might be "empRecord.empAddr[2].empAddrStreet".

The basic rules of Hashtable **loading** are:

- All data elements in the DataView will attempt to acquire a value from the Hashtable. If no matching key exists, the element will retain its original value.
- Hashtable members of the wrong type will result in ClassCastException being thrown.

Name Translator Interface Facility

A name translator interface facility is available to provide Hashtable name mappings. Both HashtableLoader and HashtableUnloader provide a constructor that accepts an argument of type "com.bea.dmd.dataview.NameTranslator". [Listing 7-6](#) shows how this interface is defined.

Listing 7-6 Name Translator Interface

```
//=====
// NameTranslator.java
//      Name Translator interface.
//
// Copyright ©2000, BEA Systems, Inc., all rights reserved.

//-----

package com.bea.dmd.dataview;
```

```
/* *****  
 * Name Translator interface.  
 * An interface for a 'functor' object that translates field names.  
 *  
 * @version $Revision: 1.1 $  
 * @author Copyright ©2000, BEA Systems, Inc., all rights reserved.  
 */  
  
    public interface NameTranslator  
    {  
        public String    translate(String input);  
    }  
  
// End NameTranslator.java
```

You can write classes that implement this interface for your application. These implementations are used to translate the key strings before the Hashtable is accessed.

Following are some useful implementations that are included in the JAM library:

Class Constructor	Purpose
NameFlattener()	Reduces the key to the portion following the final period character.
PrefixChanger(String old, String add)	Removes an old prefix & adds a new one.
PrefixChanger(String old)	Removes a prefix.

Both the `HashtableLoader` and `HashtableUnloader` classes are included in the "com.bea.dmd.dataview" package, as well as the various name translator classes.

8 Using BEA WebLogic Process Integrator with JAM

WebLogic Process Integrator is a process modeling tool and workflow environment that combines the functions of static data and server administration, workflow definition, and workflow monitoring.

WebLogic Process Integrator may be used with JAM to provide the following key features:

- Automatically routes any type of information to the right person at the right time.
- Takes full advantage of "push technology," which allows WebLogic Process Integrator to process most of an organization's work according to complex, intelligent business rules; it flags only those exceptions that require human intervention.
- Uses a breakthrough embedded workflow-enabling approach that is written in 100% pure Java.

This section discusses the following topics:

- [WebLogic Process Integrator Overview](#)
- [WebLogic Process Integrator Architecture](#)
- [Integration with JAM](#)

WebLogic Process Integrator Overview

WebLogic Process Integrator is a Java implementation of a workflow management system. A workflow management system has been defined as one that “...defines, manages, and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic.”

Such a system automates a business process by merging the sequence of activities and invoking the appropriate resources required by the various activities or steps. Individual business processes may have life cycles ranging from minutes to days (or longer), depending on the complexity and duration of the various constituent activities. To achieve this, a workflow management system must provide support in four broad functional areas:

- **Workflow definition**—Capturing the definition of the business process (workflow).
- **Workflow execution**—Managing the execution of workflow processes in an operational environment, sequencing the various activities to be performed.
- **Workflow monitoring**—Monitoring the status of workflow processes and dynamically configuring the run-time controller.
- **Data administration**—Managing organizations, users, and roles, rerouting tasks when needed, and maintaining business workflow calendars.

WebLogic Process Integrator Architecture

At the center of the WebLogic Process Integrator architecture is the Process Engine, which serves as the run-time controller and is responsible for keeping track of workflow instances and managing the execution of workflows. The high-performance WebLogic Process Integrator workflow engine has been developed using a breakthrough embedded workflow-enabling approach that is written in 100% pure Java.

The WebLogic Process Integrator data and server administration facility allows users to administer components of the WebLogic Process Integrator server and database. More specifically, it allows users to maintain functions such as adding, updating, deleting users, roles, and organizations; disconnecting users and shutting down the server, rerouting tasks from one user to another, and creating calendars that are used in workflow definition.

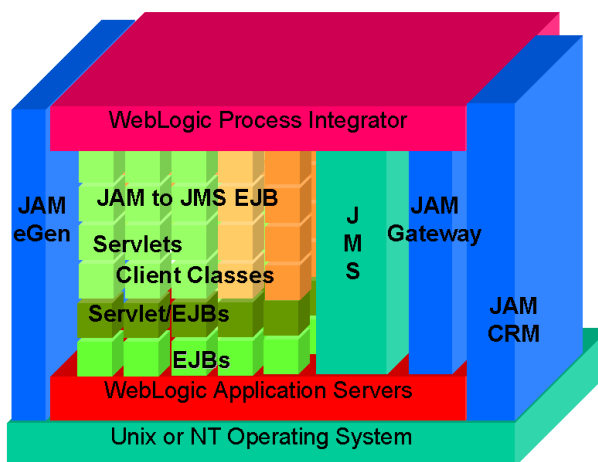
The WebLogic Process Integrator workflow definition facility allows users to graphically create and edit workflows, mapping out the tasks in the workflow and the business rules that must be followed. Workflow definitions are used as templates, which can be used to create multiple workflow instances.

The WebLogic Process Integrator workflow monitor facility allows users to monitor and modify workflows. More specifically, it allows users to perform functions such as displaying the status of workflow instances, modifying the tasks associated with a workflow instance (for example, re-assigning work tasks or forcing work to be re-done), creating a variety of customized reports on workload and statistical data, and viewing a user or role worklist in order to dynamically maintain workflows.

Finally, the WebLogic Process Integrator worklist facility provides users with a list of tasks describing work currently assigned to them (or the roles to which they belong) and provides the means for users to select and perform those tasks.

The following diagram illustrates the WebLogic Process Integrator architecture.

Figure 8-1 WebLogic Process Integrator Architecture



Integration with JAM

When using WebLogic Process Integrator with JAM, the WebLogic Process Integrator functions vary depending upon the type of request:

- **WebLogic Process Integrator to Mainframe**—Request originating in WebLogic Process Integrator sent to JAM for service in a mainframe application.
- **Mainframe to WebLogic Process Integrator**—Requests originating from the mainframe sent to JAM for service in WebLogic Process Integrator.

JAM's mechanisms for implementing both WebLogic Process Integrator to mainframe and mainframe to WebLogic Process Integrator requests have been tailored to the capabilities of WebLogic Process Integrator. WebLogic Process Integrator to mainframe requests make use of an object-based interface. Mainframe to WebLogic Process Integrator requests make use of an XML event interface.

WebLogic Process Integrator to Mainframe Requests

WebLogic Process Integrator to mainframe requests are the most frequent type of communication. In a WebLogic Process Integrator to mainframe request, a workflow requires a service from the mainframe application. The invocation of a mainframe application is essentially a two step process: the input message required by the remote application is created and the service is invoked.

JAM facilitates this interaction with the following tools:

- eGen COBOL utility

The eGen portion of JAM can generate an additional class that is required to facilitate data creation within a workflow. This additional class is generically known as a “document holder.” The document being held is the message required for the mainframe service. eGen creates a document holder as part of the code generation from the mainframe COBOL copybook.

- EJB called Application View

JAM also uses an EJB, called ApplicationView, that matches the WebLogic Process Integrator service invocation with the eGen client that packages the service request for the JAM gateway.

Using the eGen COBOL Utility for WebLogic Process Integrator to Mainframe Requests

The general process for using JAM with WebLogic Process Integrator for WebLogic Process Integrator to mainframe requests is to run the eGen COBOL utility with the mainframe COBOL copybook. This process results in the standard eGen/JAM output of java source for client and dataview. The following new outputs are provided:

- A document holder for use in WebLogic Process Integrator
- An XML Schema for use by any other application that may want to send data to JAM via XML

The compiled version of the document holder is provided to the workflow developer, while the client and dataview is deployed in the WebLogic server the way JAM clients currently are deployed. Refer to [Developing Java Applications](#) for further details on developing and deploying applications using JAM.

Workflow Development for WebLogic Process Integrator to Mainframe Requests

The document holder provides the workflow developer with methods to instantiate and populate a document object for use in a mainframe service request.

Following is the general sequence of activities for workflow development:

1. Declare business operations using the document holder that corresponds to the mainframe application to be invoked. All methods on the document holder should be declared business operations because they represent every operation required to create, populate, and extract data for the mainframe application.

When the business operations have been declared, a workflow can make reference to them. General usage of a document holder follows a standard pattern.

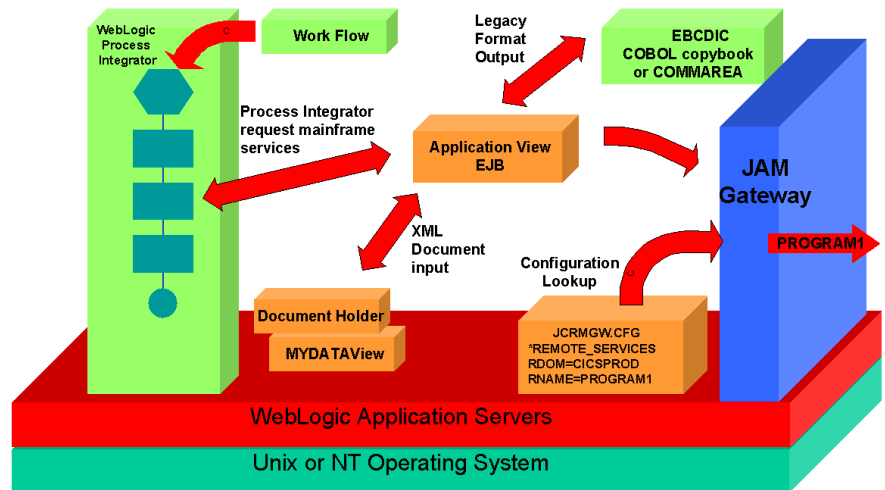
2. Use the Factory method to obtain a new instance of the document “to be held.”

3. Populate the document data elements by using the accessor business operations provided by the document holder.
4. Invoke the mainframe request using the JAM Application View providing the service name and the document. (Not the document holder)
5. Upon returning from the request, use the accessor business operations to retrieve data from the document which has been provided/updated by the mainframe application.

This general process could encompass a large number of workflow operations depending on the complexity of the data. The workflow could be manually started, in which case the factory invocation could be specified in a start node. The workflow could be initiated by an XML event where the event provided the input data to populate the document object for use in the mainframe request.

The general process for using JAM with WebLogic Process Integrator for WebLogic Process Integrator to mainframe requests is illustrated in the following diagram.

Figure 8-2 WebLogic Process Integrator to Mainframe Requests from WebLogic Process Integrator



In the above figure, the Document Holder is a lightweight class for use with WebLogic Process Integrator operations. It provides a document factory for creating new document instances. The document instance is passed in to the document holder

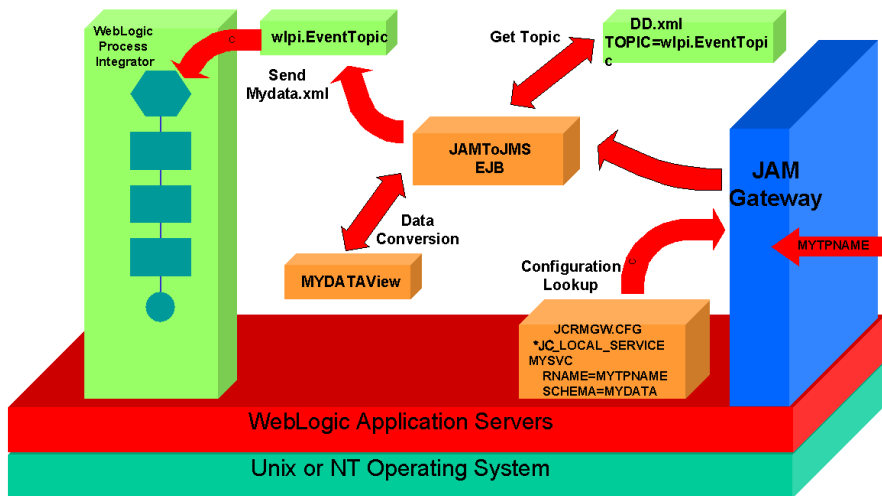
constructor when the document holder methods are used as WebLogic Process Integrator Integrator business operations. Document holder methods forward requests to actual document methods. The actual document is passed on an application view service call.

Mainframe to WebLogic Process Integrator Requests

Mainframe to WebLogic Process Integrator requests to WebLogic Process Integrator are asynchronous requests. An application used to initiate a workflow should be capable of 'request only' processing.

The general process for using JAM with WebLogic Process Integrator for Mainframe to WebLogic Process Integrator requests is illustrated in the following diagram.

Figure 8-3 Mainframe to WebLogic Process Integrator Requests to WebLogic Process Integrator



In the above figure, *MyTPName* is received from the mainframe. The local service name is obtained from the *JCRMGW.CFG* file. *JAMToJMS* is registered in JNDI with the local service name. The *JAMToJMS* EJB uses a schema name to construct a data view to convert mainframe data into XML. XML is then posted to a JMS topic found in the deployment descriptor.

Example of JAM Application Integrated with WebLogic Process Integrator

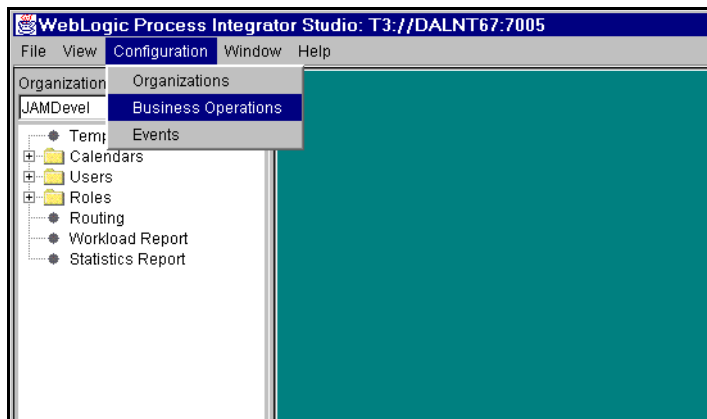
This application shows an example of JAM integration with WebLogic Process Integrator. The scenario illustrates the creation of an 'employee record' on the mainframe. This record is then read back into WebLogic Process Integrator and selected fields are extracted and stored in WebLogic Process Integrator variables. Refer to the JAM and WebLogic Process Integrator documentation for issues of operation and configuration.

This in no way is a complete illustration of WebLogic Process Integrator usage, but it does provide a starting point for constructing real workflows that interact with a JAM gateway using eGen client EJB's. Refer to your WebLogic Process Integrator documentation for more information about using WebLogic Process Integrator.

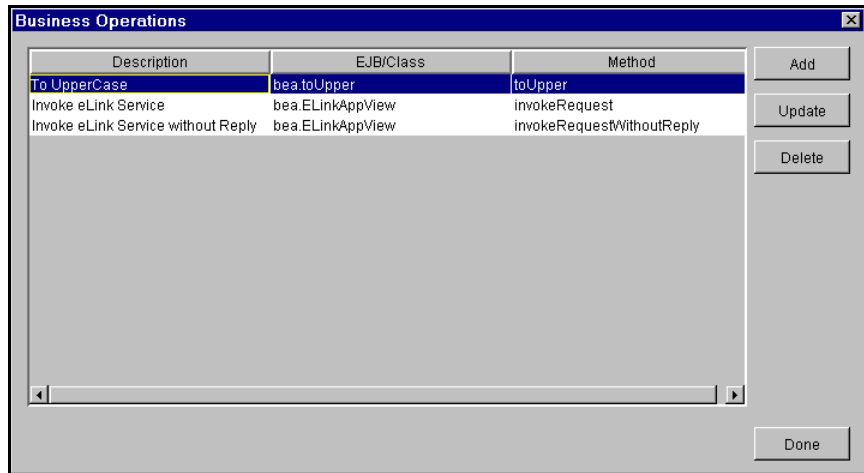
Task 1: Set Up JAM Components as Business Operations

Perform the following steps to set up JAM components as business operations:

1. Start WebLogic Process Integrator Studio
2. Choose Configuration menu → Business Operations.

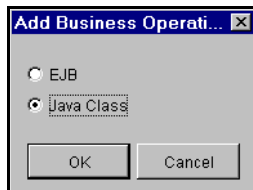


The business operation definition dialog displays.



Any existing business operations are listed. In this example business operations from the WebLogic examples are shown. The methods for the JAM Document Helper components will also be added here.

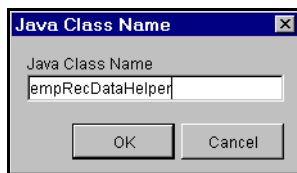
3. Click **Add** button and the following dialog displays.



The **Add Business Operation** process can accept methods from an EJB or a Java Class.

Because the JAM Data Helpers are classes, select the **Java Class** option.

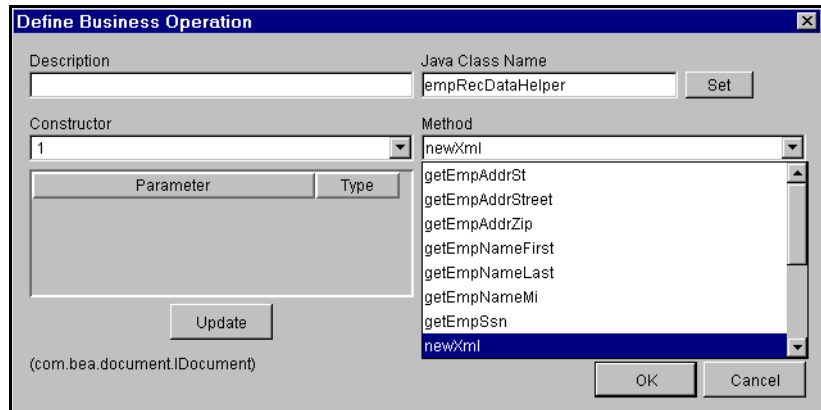
4. Click **OK** to display the Java Class Name dialog.



5. Enter the **Java Class Name** here.

This is the fully qualified class name including the complete package name as specified in the eGen script that created the data helper class. In this example, a package was not specified in the eGen script, so no package is required in this dialog. The class specified here must be located on the Java CLASSPATH. In this example the **empRecDataHelper** is located in \$WLS_HOME/myserver/clientclasses.

6. Click **OK** and the following dialog displays.



The business operation dialog has a pull-down list of all of the available methods. Methods can be selected in any order. In this example, select the Data Factory method named **newXml**. This method must be used in the workflow first to create a new **empRec** that can then be used for the other business operations.

7. Add a description for the new business operation. This description displays in the pull down list when defining a new workflow.

The dialog box titled "Define Business Operation" has a blue title bar with a close button. It contains the following fields and controls:

- Description:** A text field containing "Create New Employee Record Object".
- Java Class Name:** A text field containing "empRecDataHelper" with a "Set" button to its right.
- Constructor:** A pull-down menu showing "1".
- Method:** A pull-down menu showing "newXml".
- Parameter Table (Left):** A table with two columns: "Parameter" and "Type". It is currently empty.
- Parameter Table (Right):** A table with two columns: "Parameter" and "Type". It is currently empty.
- Buttons:** Two "Update" buttons, one below each parameter table, and "OK" and "Cancel" buttons at the bottom right.
- Footer:** The text "(com.bea.document.IDocument)" is displayed at the bottom left.

Note the left side of the dialog has a pull down list for constructors. The default constructor will always be first on this list. The newXml method must be used with the default constructor.

8. Continue adding data accessors by selecting other methods from the pull down list.

The dialog box titled "Define Business Operation" is shown in a second state with the following changes:

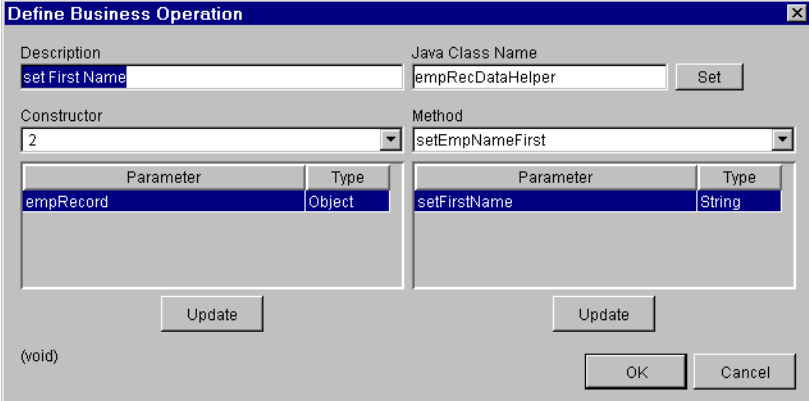
- Description:** The text field now contains "get ZipCode".
- Constructor:** The pull-down menu now shows "2".
- Method:** The pull-down menu now shows "getEmpAddrZip".
- Parameter Table (Left):** The table now contains one row: "empRecord" under the "Parameter" column and "Object" under the "Type" column. This row is highlighted with a blue background.
- Buttons:** The "Update" buttons and "OK/Cancel" buttons remain.
- Footer:** The text "(String)" is displayed at the bottom left.

The data accessors must use the second constructor provided. This constructor takes a single object as a parameter. That object is the empRec object that is created by the newXml method. This is illustrated by naming the parameter for the constructor emp Record.

Again, add a simple description to be added to the workflow definition pull-down list displayed for the business operations. In this example the `getEmpAddrZip` method is given the description `get ZipCode`.

Continue adding methods until all accessors have been set up as business operations.

Following is an example of a business operation for the `set FirstName` method. Constructor 2 is specified with the object parameter for `empRecord` and string parameter for `SetEmpNameFirst`.



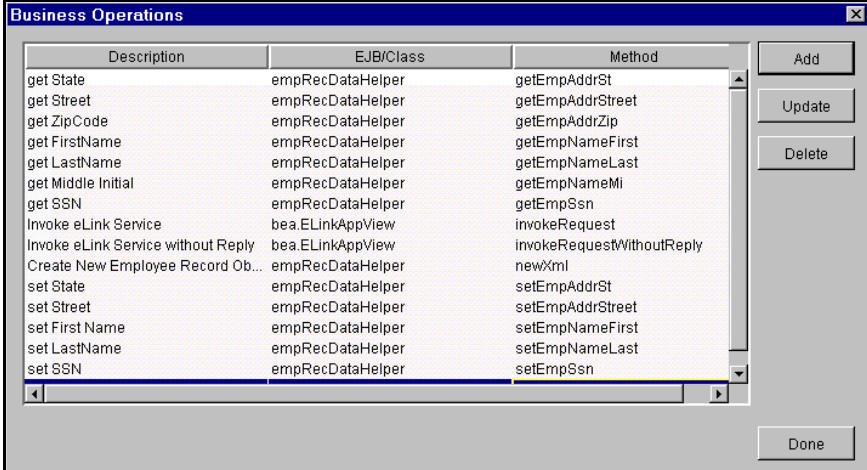
The **Define Business Operation** dialog box is shown. It contains the following fields and controls:

- Description:** `set First Name`
- Java Class Name:** `empRecDataHelper` (with a **Set** button)
- Constructor:** `2` (selected from a dropdown)
- Method:** `setEmpNameFirst` (selected from a dropdown)
- Parameter Table (Left):**

Parameter	Type
<code>empRecord</code>	<code>Object</code>
- Parameter Table (Right):**

Parameter	Type
<code>setFirstName</code>	<code>String</code>
- Buttons:** **Update** (two instances), **OK**, **Cancel**, and **(void)**.

The final list should look something like the following.



The **Business Operations** dialog box shows a list of operations. The list has three columns: **Description**, **EJB/Class**, and **Method**.

Description	EJB/Class	Method
get State	empRecDataHelper	getEmpAddrSt
get Street	empRecDataHelper	getEmpAddrStreet
get ZipCode	empRecDataHelper	getEmpAddrZip
get FirstName	empRecDataHelper	getEmpNameFirst
get LastName	empRecDataHelper	getEmpNameLast
get Middle Initial	empRecDataHelper	getEmpNameMi
get SSN	empRecDataHelper	getEmpSsn
Invoke eLink Service	bea.ELinkAppView	invokeRequest
Invoke eLink Service without Reply	bea.ELinkAppView	invokeRequestWithoutReply
Create New Employee Record Ob...	empRecDataHelper	newXml
set State	empRecDataHelper	setEmpAddrSt
set Street	empRecDataHelper	setEmpAddrStreet
set First Name	empRecDataHelper	setEmpNameFirst
set LastName	empRecDataHelper	setEmpNameLast
set SSN	empRecDataHelper	setEmpSsn

On the right side of the list are buttons: **Add**, **Update**, **Delete**, and **Done**.

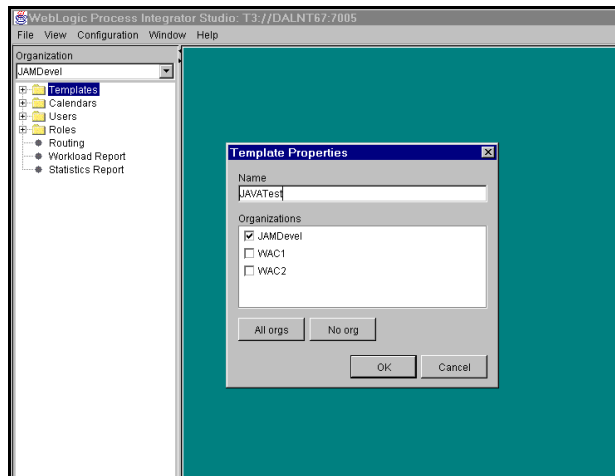
Click **Done** when you are finished and the WebLogic Process Integrator main window redisplays.

Task 2: Set Up a Workflow

Now a workflow can be defined to make use of these business operations.

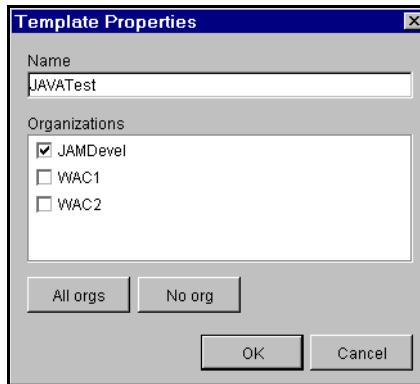
Perform the following steps to define a workflow:

1. Create a new template by selecting and right-clicking on **Templates** in the WebLogic Process Integrator Folder Tree.

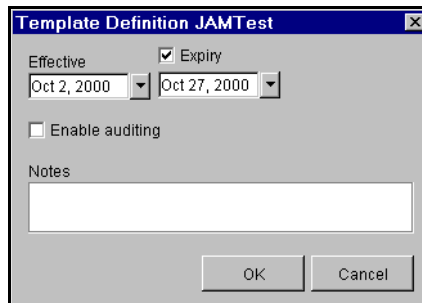


The Template Properties dialog displays.

2. Select one or more organizations for the template.



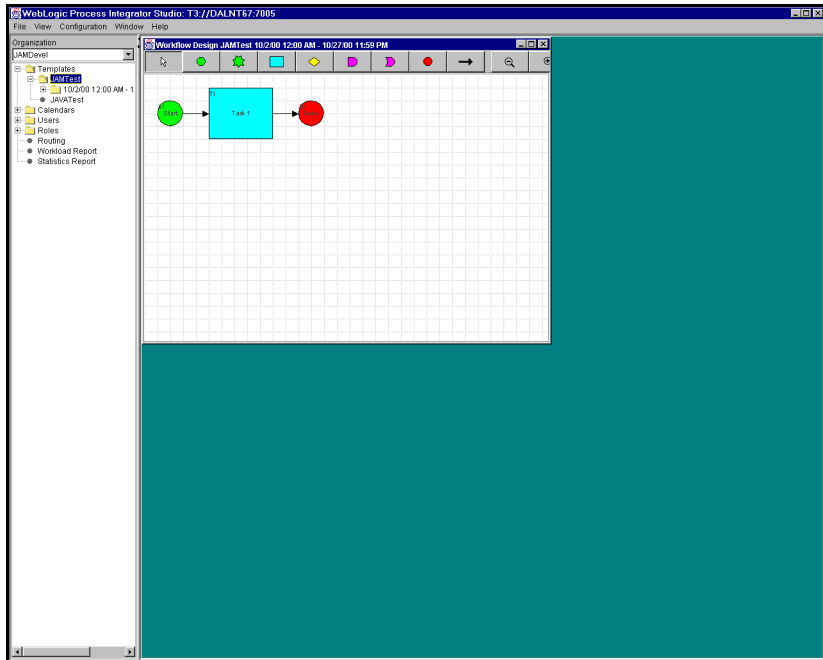
3. Right click on your new template to create a new definition and select **New Template Definition** to display the Template Definition dialog.
4. Select the characteristics for your definition.



In this example the workflow becomes effective on October 2, 2000 and expires on October 27, 2000.

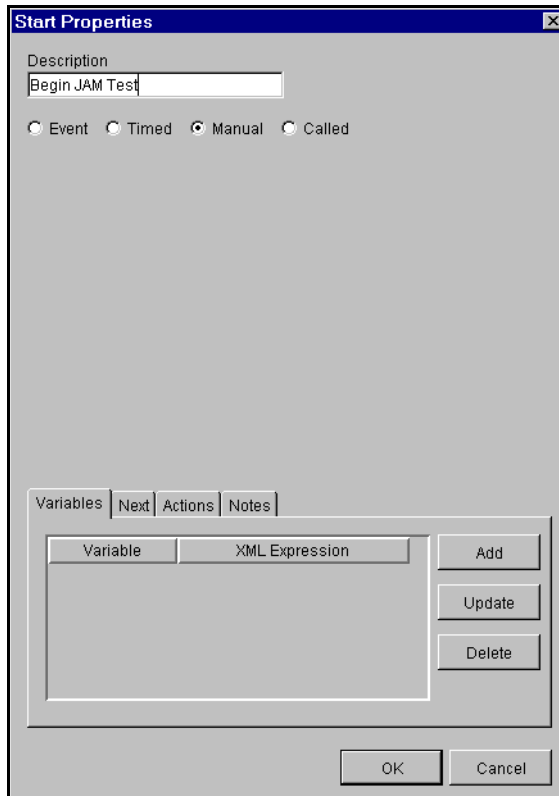
5. Click **OK**.

A new workflow opens.



Task 3: Define the Start Node

1. Double-click on the start node to begin using the empRec Data Helper. Enter *Begin JAM Test* in the **Description** text box.

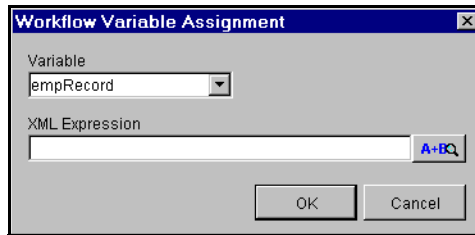


The image shows a 'Start Properties' dialog box with a blue title bar. It contains a 'Description' text box with the text 'Begin JAM Test'. Below this are four radio buttons: 'Event', 'Timed', 'Manual' (which is selected), and 'Called'. At the bottom, there are four tabs: 'Variables', 'Next', 'Actions', and 'Notes'. The 'Variables' tab is active, showing a table with two columns: 'Variable' and 'XML Expression'. To the right of the table are three buttons: 'Add', 'Update', and 'Delete'. At the very bottom of the dialog are 'OK' and 'Cancel' buttons.

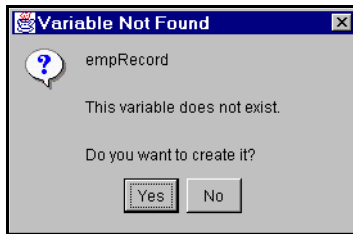
Variable	XML Expression
----------	----------------

The Start Properties dialog determines the type of workflow to be created. In this example a manually started workflow is created.

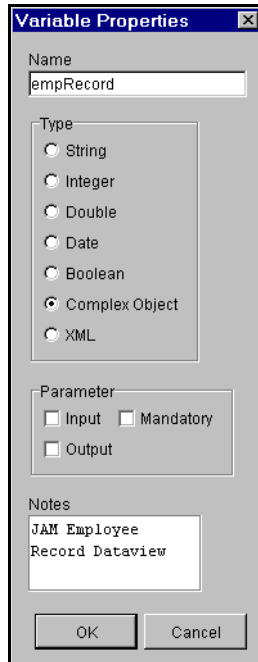
2. Click **Add** to create a variable to hold a new employee record object. The Workflow Variable Assignment dialog displays.



3. Enter the **Variable** named `empRecord` and click **OK**. Since it does not yet exist, a prompt displays asking if you want to create it.



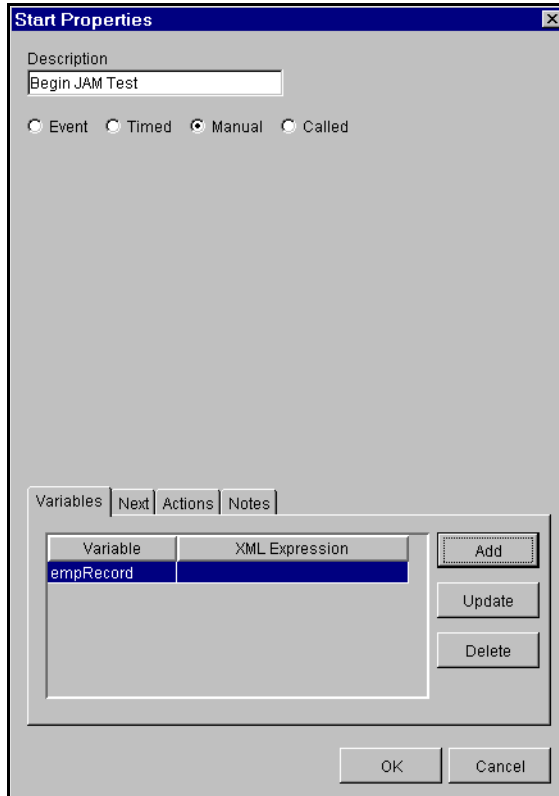
4. Click **Yes** to display the Variable Properties dialog for the new variable.



The image shows a 'Variable Properties' dialog box. It has a title bar with 'Variable Properties' and a close button. The dialog is divided into several sections. The 'Name' section has a text field containing 'empRecord'. The 'Type' section contains a list of radio buttons: String, Integer, Double, Date, Boolean, Complex Object (which is selected), and XML. The 'Parameter' section contains three checkboxes: Input, Mandatory, and Output, all of which are unchecked. The 'Notes' section has a text area containing the text 'JAM Employee Record Dataview'. At the bottom of the dialog are two buttons: 'OK' and 'Cancel'.

5. Select **Complex Object** for the **Type** because this variable will be used to hold an object reference. You may also add a description of the variable in the **Notes** text box.

6. Click **OK** and the Start Properties dialog redisplay.



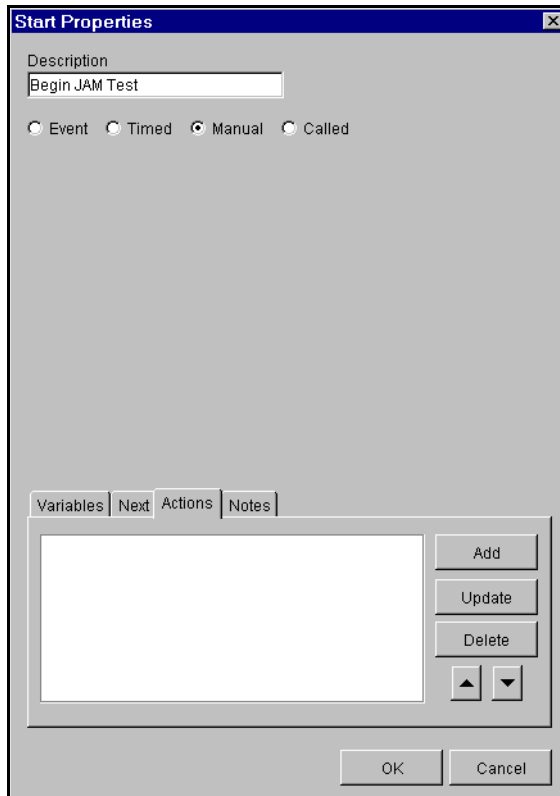
The image shows a 'Start Properties' dialog box with a blue title bar. It contains a 'Description' text field with the value 'Begin JAM Test'. Below this are four radio buttons: 'Event', 'Timed', 'Manual' (which is selected), and 'Called'. At the bottom, there are four tabs: 'Variables', 'Next', 'Actions', and 'Notes'. The 'Variables' tab is active, showing a table with two columns: 'Variable' and 'XML Expression'. The table has one row with the value 'empRecord' in the 'Variable' column. To the right of the table are three buttons: 'Add', 'Update', and 'Delete'. At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

Variable	XML Expression
empRecord	

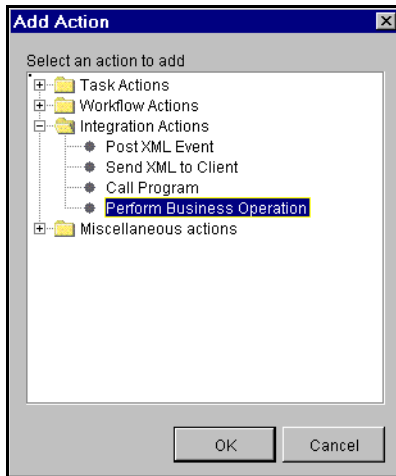
The start node is now ready to have an action added to it.

Task 4: Set Up the Start Node Properties

1. From the Start Properties dialog, select the **Actions** tab and click **Add**.

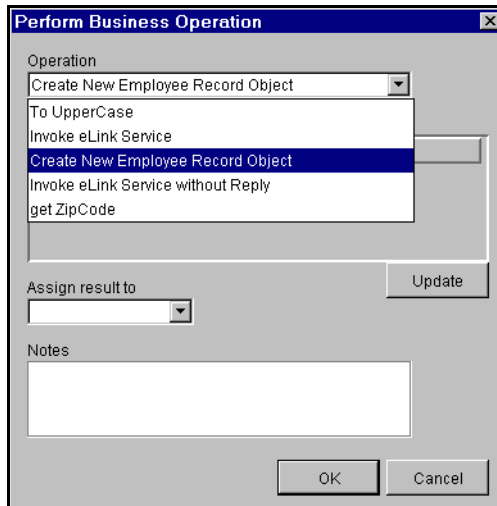


2. The Add Action dialog displays. Expand the **Integration Actions** item and select **Perform Business Operation**.

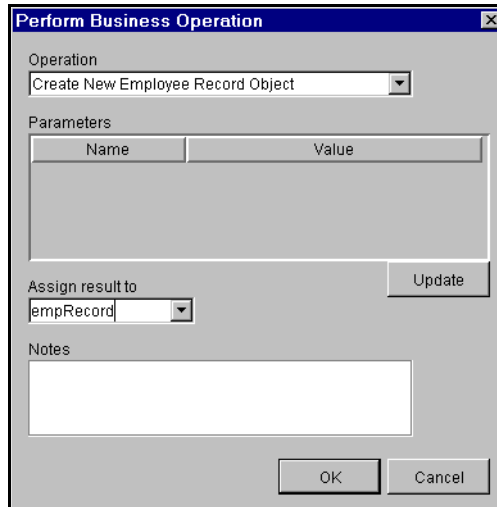


Click **OK** and the Perform Business Operation dialog displays.

3. Pull down the operation list and select the **Create New Employee Record Object** operation.



4. Pull down the **Assign result to** variable list and select the `empRecord` variable that was previously created.

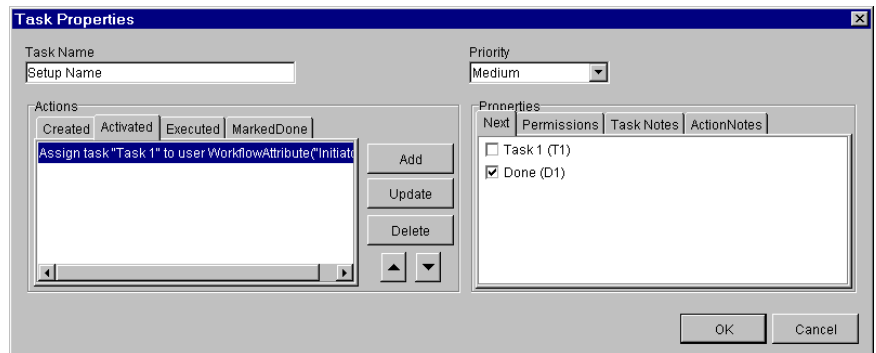


The **Perform Business Operation** dialog box is shown. It has a title bar with a close button. The **Operation** section contains a dropdown menu with the text "Create New Employee Record Object". The **Parameters** section contains a table with two columns: "Name" and "Value". Below the table is an "Assign result to" dropdown menu with "empRecord" selected. To the right of this dropdown is an "Update" button. At the bottom is a "Notes" text area. At the very bottom are "OK" and "Cancel" buttons.

This completes the start node. Click **OK** to return to the Workflow window.

Task 5: Create a Setup Name Task

1. Actions can now be added by double clicking on the Task box displayed for the workflow diagram. The Task Properties dialog displays.

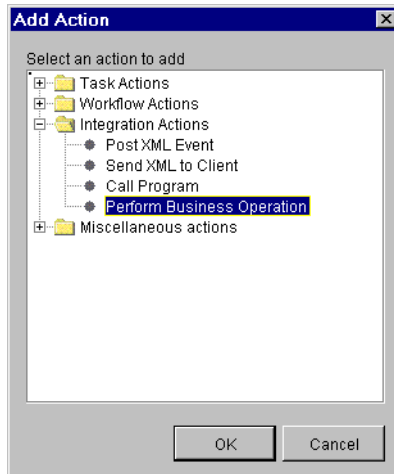


The **Task Properties** dialog box is shown. It has a title bar with a close button. The **Task Name** text field contains "Setup Name". The **Priority** dropdown menu is set to "Medium". The **Actions** section has four tabs: "Created", "Activated", "Executed", and "MarkedDone". The "Created" tab is active, showing a list of actions. The first action is "Assign task 'Task 1' to user WorkflowAttribute('Initiat...". To the right of the list are "Add", "Update", and "Delete" buttons. The **Properties** section has four tabs: "Next", "Permissions", "Task Notes", and "ActionNotes". The "Next" tab is active, showing a list of properties. The first property is "Task 1 (T1)" with an unchecked checkbox. The second property is "Done (D1)" with a checked checkbox. At the bottom are "OK" and "Cancel" buttons.

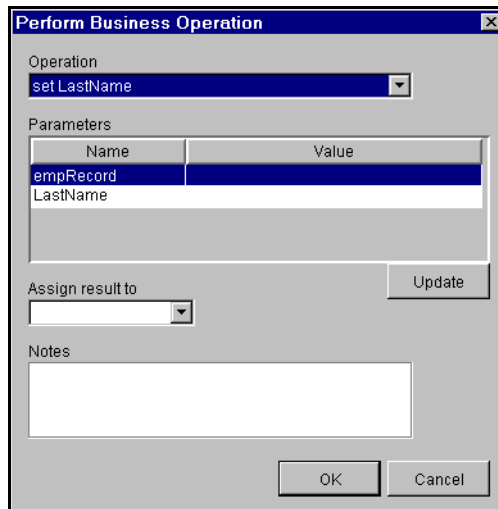
A task can use several business operations. In this example the task **Setup Name** is created and several methods are used to set values in the various fields that make up the name.

Select the **Activated** tab and click **Add** button to add a business operation on the Add Action dialog.

2. Expand the **Integration Actions**, select **Perform Business Operation**, and click **OK** to display the Perform Business Operation dialog.

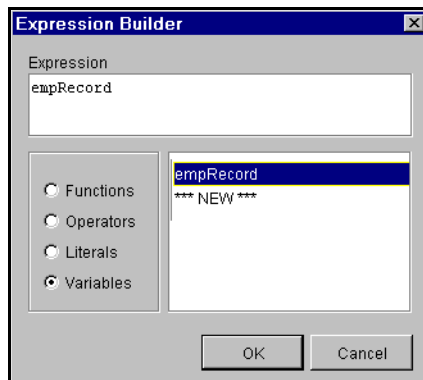


3. Select the **set LastName** operation from the pull down list in the Perform Business Operation dialog. Remember the names in this list originate from the description property on the business method as it was configured. For this reason, it is good to adopt a simple standard when creating descriptions.



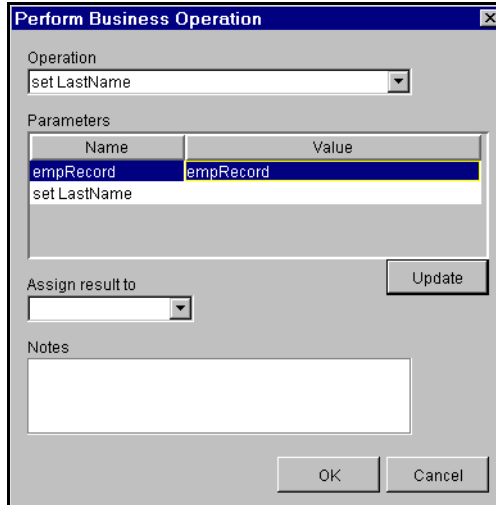
The method used to set the last name requires two parameters. One parameter is for the constructor and one parameter is for the method itself. In this example the constructor takes the **empRecord** variable that has been populated in the start node. The business operation takes a string representing the last name.

4. Double-click on the **Value** for **empRecord** to display the Expression Builder dialog.



This dialog allows you to set the value of the required parameter. In this example a variable is used. Select the **Variables** option to display a list of all the variables defined for this workflow. This example uses the **empRecord** variable that was defined and populated in the start node.

5. Select and double-click on the required variable, the variable name is set in the expression. Click **OK** to return to the Perform Business Operation dialog.



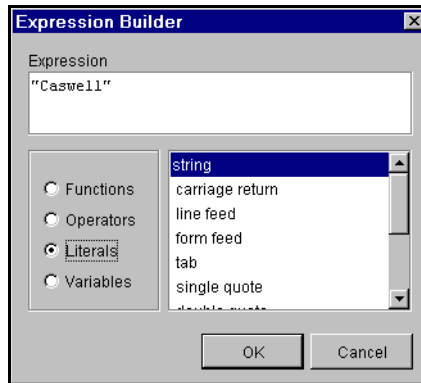
The image shows a dialog box titled "Perform Business Operation". It has a close button (X) in the top right corner. The dialog is divided into several sections:

- Operation:** A dropdown menu showing "set LastName".
- Parameters:** A table with two columns: "Name" and "Value".

Name	Value
empRecord	empRecord
set LastName	
- Assign result to:** A dropdown menu that is currently empty.
- Update:** A button located to the right of the "Assign result to" dropdown.
- Notes:** A large text area for entering notes.
- OK and Cancel:** Two buttons at the bottom right of the dialog.

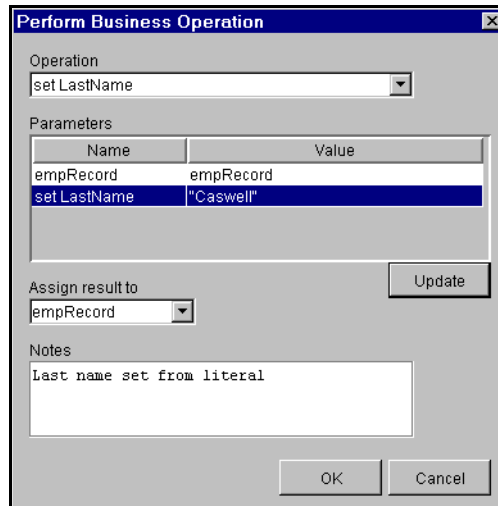
6. Select the **set LastName** operation and double-click on the **Value** for **empRecord** to display the Expression Builder dialog.

For this example, enter the last name in the **Expression** text box and select the **Literals** option. The name could have been obtained in any manner and used from a variable or other source.



Click **OK** to return to the Perform Business Operation dialog.

7. Select **empRecord** from the **Assign result to** drop-down list to complete the definition.



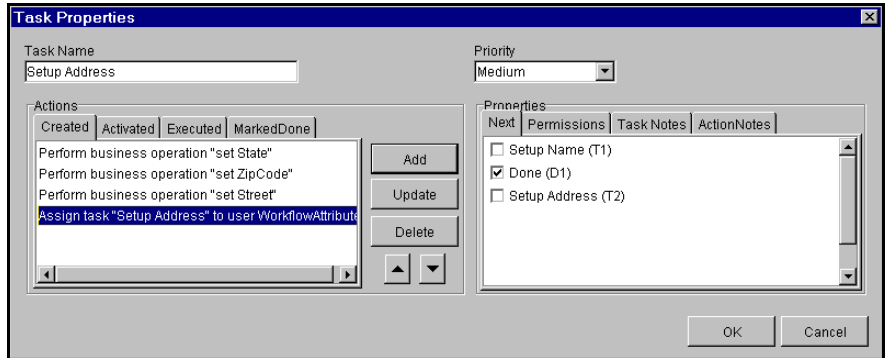
8. Click **OK** to add this business operation to the action and return to the Task Properties dialog.

The screenshot shows the 'Task Properties' dialog box. The 'Task Name' field contains 'Setup Name' and the 'Priority' dropdown is set to 'Medium'. The 'Actions' tab is active, showing a list of actions: 'Assign task "Task 1" to user WorkflowAttribute("Initiat' and 'Perform business operation "set LastName"'. The 'Properties' tab is also visible, showing a list of properties: 'Task 1 (T1)' and 'Done (D1)'. The 'Done (D1)' property is checked. The 'OK' and 'Cancel' buttons are at the bottom right.

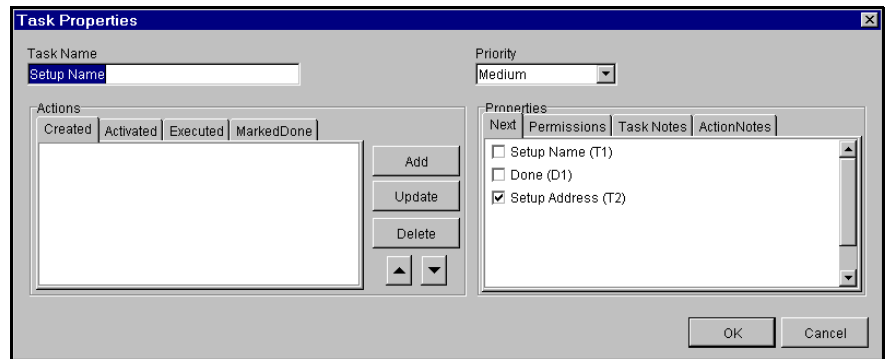
9. Repeat steps through 8. to add the additional business operations required to set up the employee record. These operations will include **setFirstName** and **setMiddleName**.

The screenshot shows the 'Task Properties' dialog box after adding more actions. The 'Task Name' field contains 'Setup Name' and the 'Priority' dropdown is set to 'Medium'. The 'Actions' tab is active, showing a list of actions: 'Assign task "Setup Name" to user WorkflowAttribute("",', 'Perform business operation "set LastName"', 'Perform business operation "set First Name"', 'Perform business operation "To UpperCase"', and 'Perform business operation "set Middle Initial"'. The 'Properties' tab is also visible, showing a list of properties: 'Setup Name (T1)' and 'Done (D1)'. The 'Done (D1)' property is checked. The 'OK' and 'Cancel' buttons are at the bottom right.

10. Create an additional Task named **Setup Address** and link it to the **Done** node.



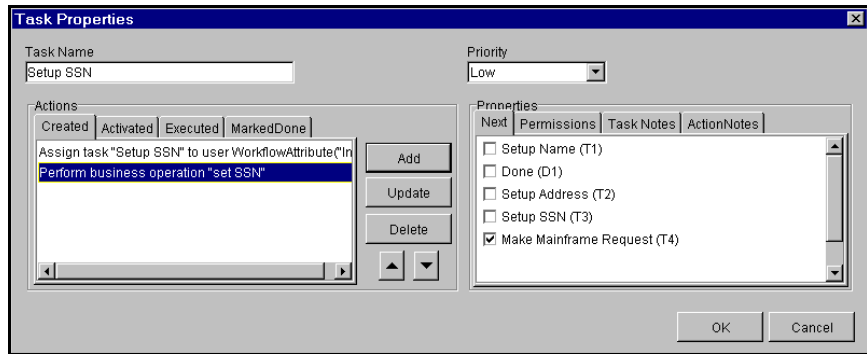
11. Link the **Setup Name** task to the **Setup Address** task by selecting **Setup Address** in the **Next** property.



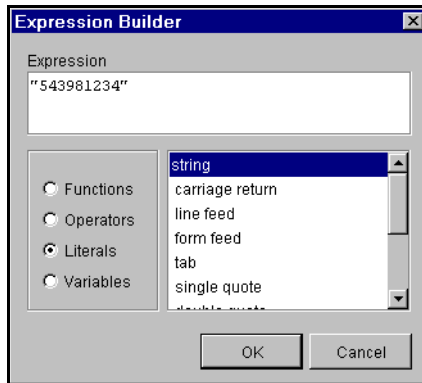
Task 6: Create Additional Tasks

This section shows examples that were set up similarly to the tasks described in the previous steps. Refer to your WebLogic Process Integrator documentation for more information about using WebLogic Process Integrator Studio.

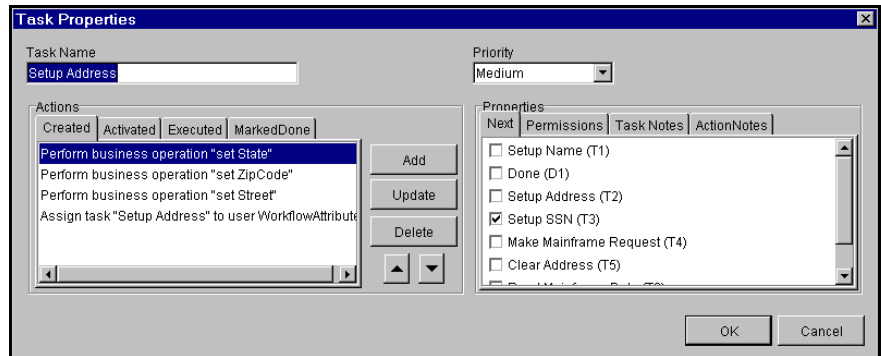
The following figure shows the example from the previous steps with a new task named **Setup SSN** added to set the social security number.



All of the DataHelper accessor set methods take string parameters. The data conversion is handled automatically for the field. Therefore, in this example the value set for the social security **Expression** is a literal string.



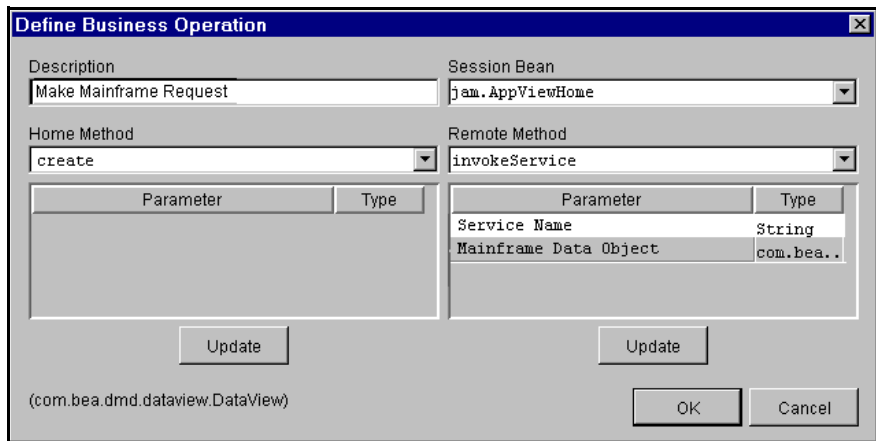
The **Setup Address** task is linked to the **Setup SSN** task.



Since the employee record only requires a name, address, and social security, a **Make Mainframe Request** task can now be created.

Once the data has been set up, the mainframe request can be made. All JAM requests are made through the JAM Application View **invokeService** method. The JAM Application view is a stateless session enterprise Java bean registered in JNDI as `jam.AppViewHome`. The actual client EJBs used to invoke the service do not need to be known to WebLogic Process Integrator. The AppView will invoke the proper bean and method based upon the service name. The service name must match a remote service that has been configured for the JAM gateway. JAM gateway configuration is specified in the `jcrmgw.cfg` file.



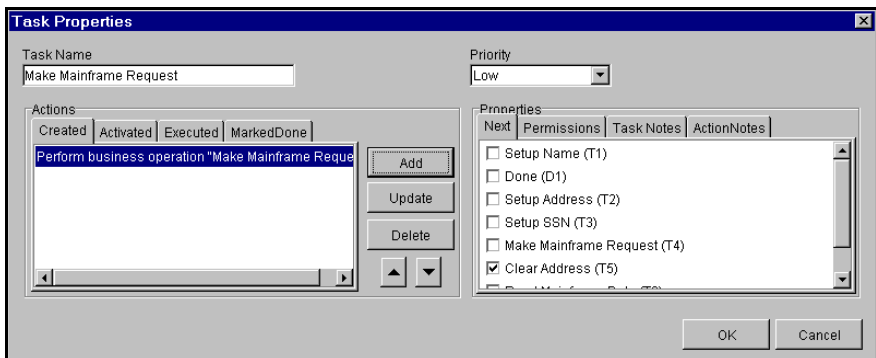


The **Define Business Operation** dialog box is used to configure a business operation. It contains the following fields and controls:

- Description:** A text field containing "Make Mainframe Request".
- Session Bean:** A dropdown menu showing "jam.AppViewHome".
- Home Method:** A dropdown menu showing "create".
- Remote Method:** A dropdown menu showing "invokeService".
- Parameter Table:** A table with two columns: "Parameter" and "Type". It contains two rows:

Parameter	Type
Service Name	String
Mainframe Data Object	com.bea..
- Update Buttons:** Two "Update" buttons, one for each parameter table.
- Footer:** A text field containing "(com.bea.dmd.dataview.DataView)" and "OK" and "Cancel" buttons.

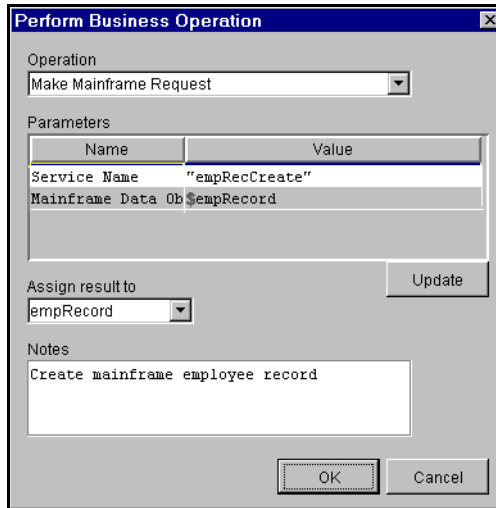
The business operation required is **invokeService**, which will use the default constructor and take two parameters. The first parameter is a **Service Name** and the second is the **Mainframe Data Object** that was created with the **newXml** method invoked in the Start node.



The **Task Properties** dialog box is used to configure a task. It contains the following fields and controls:

- Task Name:** A text field containing "Make Mainframe Request".
- Priority:** A dropdown menu showing "Low".
- Actions:** A list box containing "Perform business operation 'Make Mainframe Request'". It has "Add", "Update", and "Delete" buttons.
- Properties:** A tabbed section with "Next", "Permissions", "Task Notes", and "ActionNotes" tabs. The "ActionNotes" tab is selected, showing a list of actions:
 - ☐ Setup Name (T1)
 - ☐ Done (D1)
 - ☐ Setup Address (T2)
 - ☐ Setup SSN (T3)
 - ☐ Make Mainframe Request (T4)
 - ☒ Clear Address (T5)
- Footer:** "OK" and "Cancel" buttons.

A new task named **Make Mainframe Request** is created using the **invokeService** business operation.



Perform Business Operation

Operation: Make Mainframe Request

Parameters:

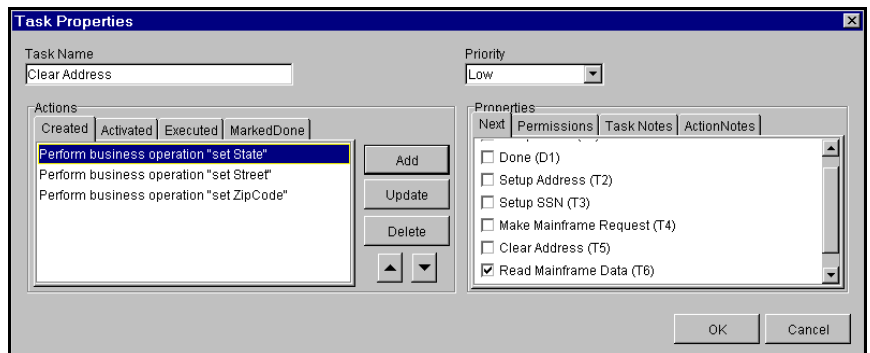
Name	Value
Service Name	"empRecCreate"
Mainframe Data Ob	\$empRecord

Assign result to: empRecord

Notes: Create mainframe employee record

OK Cancel

For purposes of this example, a **Clear Address** task was created so that the workflow can read the item just created.



Task Properties

Task Name: Clear Address

Priority: Low

Actions:

Created	Activated	Executed	MarkedDone

Perform business operation "set State"

Perform business operation "set Street"

Perform business operation "set ZipCode"

Add Update Delete

Properties:

Next	Permissions	Task Notes	ActionNotes

Done (D1)

Setup Address (T2)

Setup SSN (T3)

Make Mainframe Request (T4)

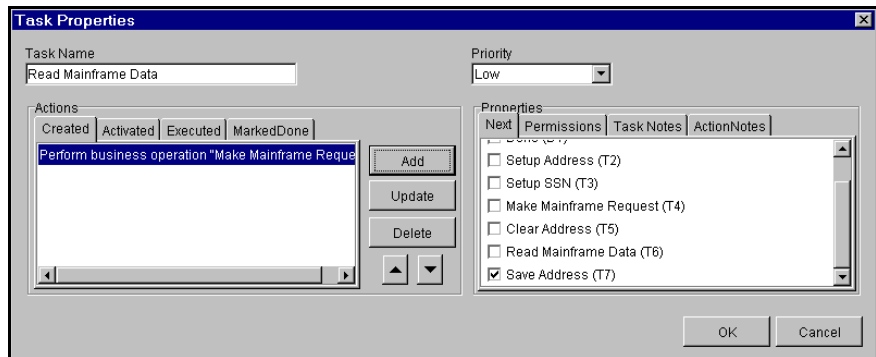
Clear Address (T5)

Read Mainframe Data (T6)

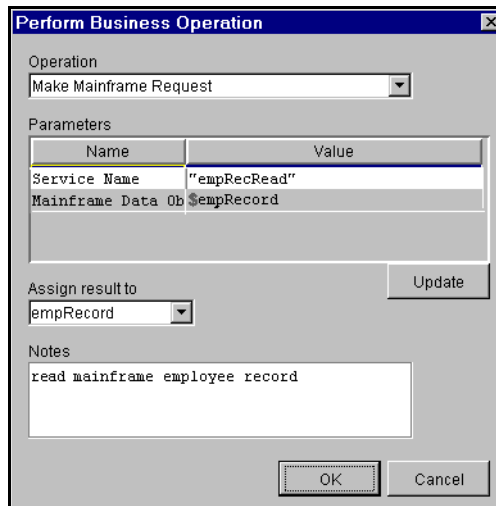
OK Cancel

The **set** methods are used to clear state, street and zip code.

A new task called **Read Mainframe Data** has been created and linked to a task called **Save Address** that will extract state, street and zip code.

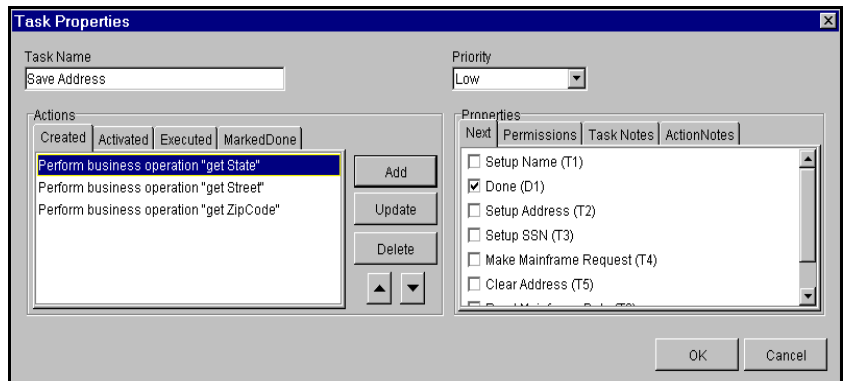


The business operation, **Make Mainframe Request**, is again used but in this example the service name is **empRecRead**. This service has been set up in the `jcrmgw.cfg` as a remote service.



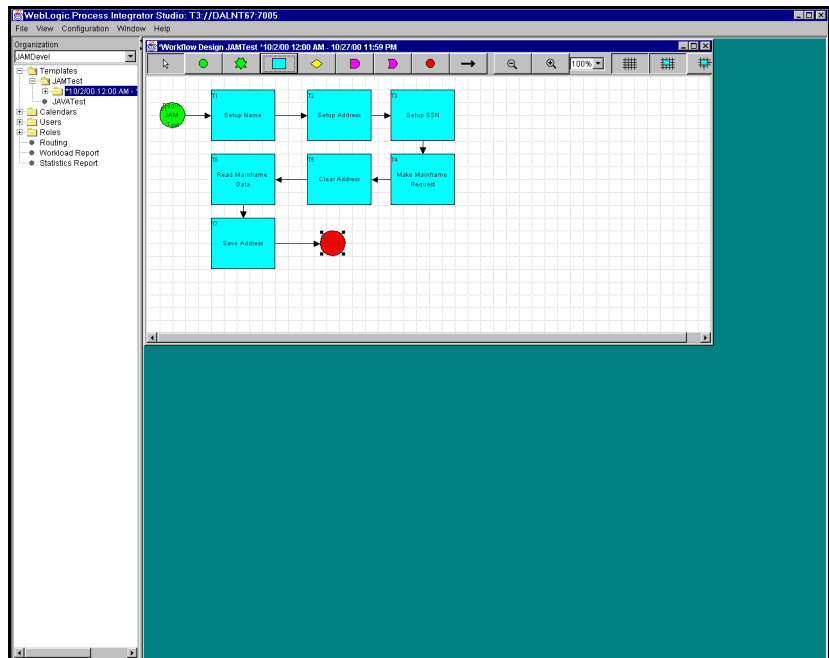
After reading the mainframe employee record the address fields will be extracted into WebLogic Process Integrator variables.

In the following example, three new variables for state, street and zip code have been added.



The task, **Save Address** uses the **get** methods to extract the state, street, and zip code that was returned from the mainframe. This task is linked to a **Done** node, however additional processing could have been added.

Following is an example of how the finished workflow might look.



9 Developing a Multi-Service Data Entry Servlet

This section contains a scenario that shows how to develop a multi-service application, as opposed to the single-service application presented in [Generating a Servlet-Only JAM Application](#). This scenario is based on the general procedures presented in [Chapter 4, “Developing Java Applications,”](#) It gives you practical examples for using JAM tools, presented as tasks with step-by-step procedures. This scenario depicts the development of a new application and the updating of existing applications. 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.

The following tasks are required to develop a multi-service application:

- [Task 1: Use eGen COBOL to Create a Base Application](#)
- [Task 2: Create Your Custom Application from the Base Application](#)
- [Task 3: Update the JAM Configurations and Update BEA WebLogic Server Properties](#)
- [Task 4: Deploy Your Application](#)
- [Task 5: Use the Application](#)

The following section describes the sample programs required for the multi-service application:

- [Sample COBOL Programs for the Form Buttons](#)

Task 1: Use eGen COBOL to Create a Base Application

You must first 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 `emprec.cbl`, as shown in [Listing 9-1](#).

Listing 9-1 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

In [Listing 9-2](#), the data view `emprecData` is generated from the copybook named `emprec.cbl`.

Listing 9-2 Basic eGen COBOL script

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

Step 2: Add Service Entries

Add the single line service entries in [Listing 9-3](#) for create, read, update, and delete operations. They all use `empRecData` as input and return `emprecData` as output. In this example, a single data view is used; however, the input and output data views could be different.

Listing 9-3 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 9-4](#) associate buttons on the HTML display with services declared in the previous step.

Listing 9-4 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 9-5](#), `empRecServlet` is the servlet name to be registered as a URL in the WLS properties file. (Every servlet requires a URL to be registered this way. Refer to WLS documentation about deploying servlets for more specific information.) Here, the `empRecPage` is to be displayed when the `empRecServlet` is invoked.

Listing 9-5 Add Servlet Name

```
servlet empRecServlet shows empRecPage
```

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

Step 5: Generate the Java Source Code

In [Listing 9-6](#), invoke the eGen COBOL code generator to create the base application that is then compiled. This makes class files (`*.class`) available for servlet customizing. The `empRecData.java` is the data view object for `emprec.cbl`.

Warning: `CLASSPATH` should include the WLS subdirectories 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 9-6 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

It is a good idea to obtain a list of accessors for use later. In general, you should 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 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 the [Listing 9-7](#), the output from the `grep` command shows the relationships in reverse order, for example:

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

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

Listing 9-7 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 Base Application

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

Step 1: Start with Imports

In [Listing 9-8](#), `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 9-8 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 9-9](#) shows how to extend the generated servlet. This enables regeneration of the base application without destroying customized code. Fields can be added to the copybook without disrupting the customized code.

Listing 9-9 Declaring the New Custom Class

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

:
:

Step 3: Add Implementation for doGetSetup

In [Listing 9-10](#), you can see how to provide a new data view and the http session. The `HttpSession (s)` can be used to hold a reference to the data view. This ensures you are actually in the first pass rather than a browser back arrow. The data view provided (`dv`) is a fresh instance of the `empRecData` data view.

Listing 9-10 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
}
```

Step 4: Continue Implementation for doGetSetup

In [Listing 9-11](#), note the use of group level accessors to obtain fields. This code pre-fills fields with data entry hints as to what fields are required, or how numeric values should be entered. You can fill form data in any manner required prior to displaying.

Listing 9-11 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");
```

Step 5: Finish Implementation for doGetSetup

In [Listing 9-12](#), note the use of the `http session putValue` to save a reference to the data view. The `doGet()` processing continues on return. This data will be presented in the displayed form.

Listing 9-12 Finish Implementation for doGetSetup

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

Step 6: Create Implementation for doPostSetup

In [Listing 9-13](#), the data view passed in contains values entered into the form by the application user. (The `HttpSession` is also available for use at this point, if required.)

Listing 9-13 Create Implementation for doPostSetup

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

Step 7: Continue Implementation for doPostSetup

In [Listing 9-14](#), note the use of group-level accessors to obtain fields. This code checks for original defaults, as well as missing data. The `SocialSecurity` is a `BigDecimal` object. Validation can be simple or complex as necessary.

Listing 9-14 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(123121234L)) == 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().compareTo("Entry
    Required") == 0)
    throw new Error("Last Name Is Required");
```

Step 8: Continue Implementation of doPostSetup

In [Listing 9-15](#), 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 9-15 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");
```

Step 9: Continue Implementation for doPostSetup

In [Listing 9-16](#), 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 enables routines to be developed and tested with a servlet and easily re-used in an EJB.

Listing 9-16 Continue Implementation for doPostSetup

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

Step 10: Finish Implementation of doPostSetup

In [Listing 9-17](#), the http session `getValue` is used to remove a reference to the data view. This prevents re-posting the same data twice. The `doPost` processing continues on return. This data is now passed to the mainframe.

Listing 9-17 Finish Implementation for doPostSetup

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

Step 11: Create Implementation for doPostFinal

In [Listing 9-18](#), 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 9-18 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 12: Update the jcrmgw.cfg File with Service Entries

[Listing 9-19](#) defines 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 9-19 Update jcrmgw.cfg File

<code>empRecCreate</code>	<code>RDOM="CICS410"</code> <code>RNAME="DPLDEMOC"</code>
<code>empRecRead</code>	<code>RDOM="CICS410"</code> <code>RNAME="DPLDEMOR"</code>
<code>empRecUpdate</code>	<code>RDOM="CICS410"</code> <code>RNAME="DPLDEMOU"</code>
<code>empRecDelete</code>	<code>RDOM="CICS410"</code> <code>RNAME="DPLDEMOD"</code>

Step 13: Create Basic Three-Part HTML Frame

In [Listing 9-20](#), 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 9-20 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 9-21](#) 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 9-21 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 Properties

Update the `jcrmgw.cfg` file with the remote service entries shown in [Listing 9-22](#). The Java gateway must be restarted for new services. The entries are used when the corresponding form button is pushed. The Create button triggers `empRecCreate`, which invokes `DPLDEMOC`. The service name must match values in the `eGen COBOL` script. In this example, the `RNAME` must match an actual CICS program name.

Listing 9-22 Remote Service Entries for Create/Read/Update/Delete

<code>empRecCreate</code>	<code>RDOM="CICS410"</code> <code>RNAME="DPLDEMOC"</code>
<code>empRecRead</code>	<code>RDOM="CICS410"</code> <code>RNAME="DPLDEMOR"</code>
<code>empRecUpdate</code>	<code>RDOM="CICS410"</code> <code>RNAME="DPLDEMOU"</code>

```
empRecDelete      RDOM="CICS410 "  
                  RNAME="DPLDEMOD"
```

Update the `weblogic.properties` file with the entries shown in the [Listing 9-23](#).

Listing 9-23 Update WLS Properties File

```
weblogic.httpd.register.customEmpRec=customCrud
```

Task 4: Deploy Your Application

At this point, you have a basic form capable of receiving data entry, along with some static HTML code for display. The following are standard WLS servlet deployment steps:

1. If required, compile the *.java files.
2. Copy the class files to the WLS servlet directory.
3. If required, add URL registration lines to the `weblogic.properties` file.
4. If required, restart WLS.
5. Test the result with your browser pointed at the registered URL.

Task 5: Use the Application

Figure 9-1 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 9-1 New Data Entry Servlet Display

The screenshot shows a Netscape browser window titled "eGen - Netscape". The address bar displays "http://dalibm2.beesys.com:7001/egen/emprec/egen.html". The main content area displays a web form titled "empRecord". The form is organized into sections: "empSsn" with a text field containing "123121234"; "empName" with sub-fields "empNameLast" (containing "Entry Required"), "empNameFirst" (containing "Entry Required"), and "empNameMi" (empty); and "empAddr" with sub-fields "empAddrStreet" (containing "Entry Required"), "empAddrSt" (containing "TX"), and "empAddrZip" (containing "123451234"). At the bottom of the form are buttons for "Create", "Read", "Update", "Delete", "Clear Form", and "Refresh Form". On the left side of the browser window, there is a sidebar with a "Built On bea WEBLOGIC" logo and a section titled "empRec Info" containing links for "Introduction", "EmpRec", "Create", "Read", "Update", and "Delete". The status bar at the bottom of the browser window shows "Document Done".

Figure 9-2 shows the servlet with the Create HELP page displayed in a new window over the application.

Figure 9-2 Servlet with HELP Page Displayed

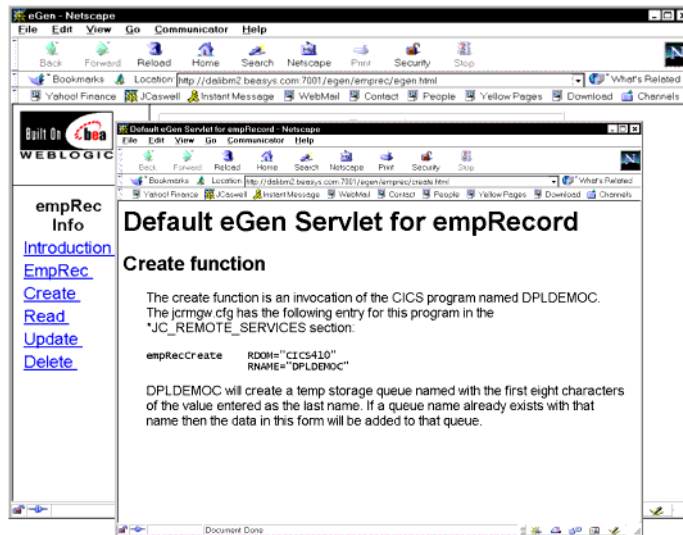
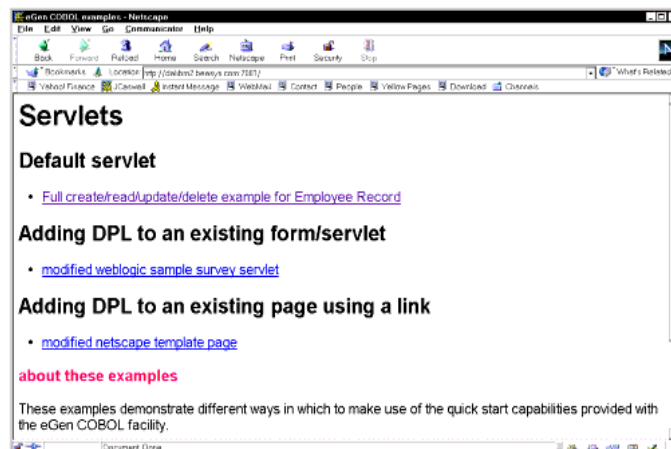


Figure 9-3 is an example of the page used for the front end of the new custom servlet.

Figure 9-3 New Data Entry Servlet Front End Page



Sample COBOL Programs for the Form Buttons

The following listings show COBOL programs for each of the button and service combinations:

- [Create](#) (DPLDEMOC)
- [Read](#) (DPLDEMOR)
- [Update](#) (DPLDEMOU)
- [Delete](#) (DPLDEMOC)

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

Create

The simple program shown in [Listing 9-24](#) writes a temporary storage queue using the first eight characters of the employee name as the QID.

Listing 9-24 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 9-25](#) 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 in consideration of the client application so residual data does not appear as the result of a read.

Listing 9-25 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 9-26](#) 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 9-26 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 9-27](#) deletes a temporary storage queue using the first eight characters of the employee name as the QID. The COMMAREA is reset in consideration of the client application so residual data does not remain after the delete.

Listing 9-27 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.
```

10 Enhancing an Existing Servlet to Originate a Mainframe Request

This scenario illustrates how to enhance an existing servlet to originate a mainframe request. Use the WebLogic Server `survey` servlet and add a mainframe request to the `post` routine. You add the code to the `postprocessing` routine, creating a mainframe buffer and sending it CICS where an application writes the buffer to a temporary storage queue and returns.

This scenario is based on the general procedures presented in [“Developing Java Applications.”](#) It gives practical examples for using JAM tools, presented as tasks with step-by-step procedures. This scenario depicts the development of a new application and the updating of existing applications. 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.

The following tasks are required to develop a multi-service application:

- [Task 1: Use eGen COBOL to Create a Base Class](#)
- [Task 2: Update the Survey Servlet Using the Generated Class](#)
- [Task 3: Update the JAM Configurations and Update WebLogic Server Properties](#)

- [Task 4: Deploy Your Application](#)
- [Task 5: Use the Application](#)

The following section describes the sample program required for the mainframe request application:

- [Sample COBOL Program to Write to Temporary Storage Queue](#)

Task 1: Use eGen COBOL to Create a Base Class

You should have successfully created the `survey` servlet prior to attempting the enhancement 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 `survey.cbl`, shown in [Listing 10-1](#).

Listing 10-1 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 COBOL Script

In [Listing 10-2](#), both the data view `surveyData` and the client class `SurveyClient` are generated from the copybook `survey.cbl`.

Listing 10-2 Basic eGen COBOL 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

In [Listing 10-3](#), you invoke the eGen COBOL code generator to create the base class that is then compiled. This makes class files (*.class) available for servlet customizing. The `surveyData.java` is the data view 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 10-3 Generating the Java Source Code

```
egencobol survey.egen
ls *.java
SurveyServlet.java surveyData.java SurveyClient.java
javac *.java
```

Step 3: Review the Java Source Code

It is a good idea to obtain a list of accessors for use later. In general, you should look at the eGen COBOL output to become familiar with each of the scenarios presented in this section.

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 10-4](#), the output from the `grep` command shows the relationships in reverse order, for example:

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

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

Listing 10-4 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 2: Update the Survey Servlet Using the Generated Class

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

Step 1: Start with Imports

In [Listing 10-5](#), `bea.jam.egen` provides the `eGen` COBOL client and data view base. The `surveyData` is the specific data view generated from the COBOL copybook. `SurveyClient` is the generated client class.

Listing 10-5 Using Imports to Start Creating the Custom Application

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

Step 2: Add New Data Members

In [Listing 10-6](#), 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 10-6 Adding New Data Members

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

Step 3: Update doPost with Mainframe Request

In [Listing 10-7](#), add the local variables for form data and data view in `doPost`. The data view is the minimum requirement. The `values` entry has been declared previously.

Listing 10-7 Update doPost with Mainframe Request

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

Step 4: Continue Updating doPost by Extracting Form Data

In [Listing 10-8](#), the code loops through the form using data view 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 10-8 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 10-9](#), 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 data view that contains any response.

Listing 10-9 Continue Updating doPost

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

The `snaException` is the base class for several exceptions, shown in [Listing 10-10](#). A time-out is the most likely error an application would get.

Listing 10-10 Mainframe Exceptions

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

Task 3: Update the JAM Configurations and Update WebLogic Server Properties

In [Listing 10-11](#), 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 10-11 Update the `jcrmgw.cfg` File with Service Name

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

Update the `weblogic.properties` file with the entries shown in [Listing 10-12](#).

Listing 10-12 Update WebLogic Server Properties File

```
weblogic.httpd.register.survey=examples.servlets.SurveyServlet
```

Task 4: Deploy Your Application

At this point, you have a basic form capable of making a maintenance request. The following are standard WebLogic Server servlet deployment steps:

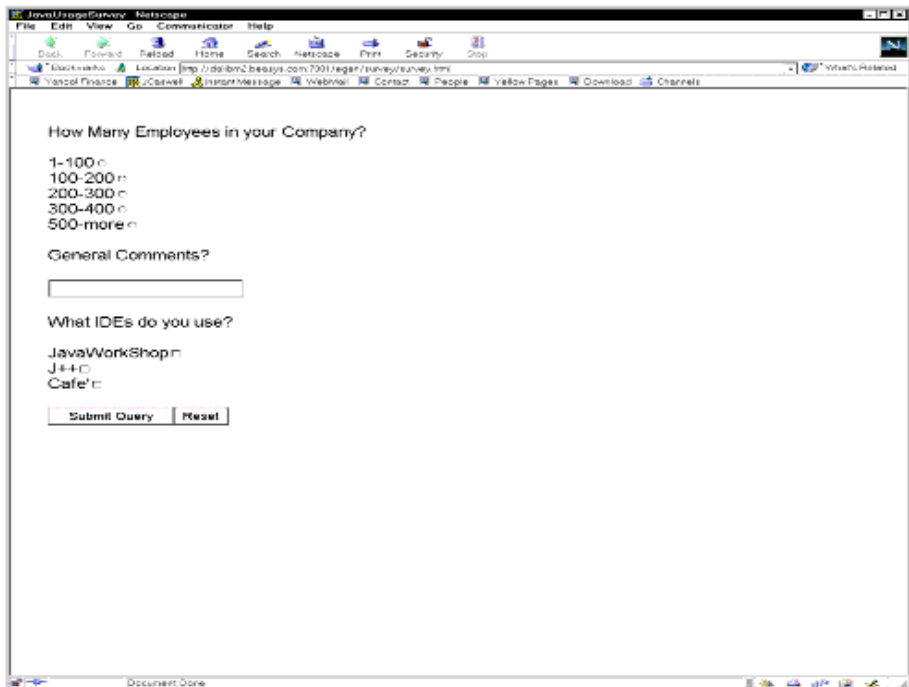
1. If required, compile *.java files.
2. Copy class files to the WebLogic Server servlet directory.
3. Copy HTML form to WebLogic Server document directory.
4. If required, add URL registration to `weblogic.properties` file.

5. If required, restart WebLogic Server.
6. Test the result with your browser pointed at the registered URL.

Task 5: Use the Application

Figure 10-1 shows the HTML display of the enhanced application.

Figure 10-1 Enhanced Survey Servlet Display



The screenshot shows a Netscape browser window with the title 'JavaUsageSurvey'. The address bar displays 'http://delim2.bea.com:7001/agent/usage/survey.html'. The survey form contains the following elements:

- Question: 'How Many Employees in your Company?'
- Radio button options:
 - 1-100
 - 100-200
 - 200-300
 - 300-400
 - 500-more
- Text input field labeled 'General Comments?'
- Text input field labeled 'What IDEs do you use?'
- Radio button options:
 - JavaWorkShop
 - J++
 - Cafe'
- Buttons: 'Submit Query' and 'Reset'

Sample COBOL Program to Write to Temporary Storage Queue

The simple program shown in [Listing 10-13](#) 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 10-13 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.

11 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. In this scenario, 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.

This scenario is based on the general procedures presented in [Chapter 4, “Developing Java Applications,”](#) It gives you practical examples for using JAM tools, presented as tasks with step-by-step procedures. This scenario depicts the development of a new application and the updating of existing applications. 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.

The following tasks are required to update and existing EJB to service a request from the mainframe:

- [Task 1: Use eGen COBOL to Create a Base Class](#)
- [Task 2: Update the Trader Interface Using the Generated Class](#)
- [Task 3: Update the JAM Configurations](#)

- [Task 4: Deploy Your Application](#)
- [Task 5: Use the Application](#)

The following section describes the sample programs required for updating an EJB to service a mainframe request:

- [Sample COBOL Program to Write to Temporary Storage Queue](#)

Task 1: Use eGen COBOL to Create a Base Class

You should have successfully run the WebLogic Server basic `statelessSessionTraderBean` 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 11-1](#).

Listing 11-1 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 11-2](#) generates the data view `traderData` from the copybook named `trader.cbl`. The script is then saved as `inboundEJB.egen`.

Listing 11-2 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 11-3](#), you invoke the eGen COBOL code generator to compile `trader.cbl` copybook and `inboundEJB.egen`. The `traderData.java` is the data view 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 11-3 Generating the Java Source Code

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

Step 3: Review the Java Source Code

It is a good idea to 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 deriving the output from generated code. The base application can be regenerated without destroying the custom code.

11 *Updating an Existing EJB to Service a Mainframe Request*

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 11-4](#), the output from the `grep` command shows the relationships in reverse order, for example:

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

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

Listing 11-4 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

You are now ready to update the WebLogic Server trader example basic statelessSession bean.

Step 1: Start with Import

In [Listing 11-5](#), 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 11-5 Using Imports to Start Updating the EJB

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

Step 2: Continue with Imports

In [Listing 11-6](#), you perform four imports to update the EJB. The `bea.base.io.*` import provides the mainframe reader and writer. The `traderData` import is the specific data view generated from the COBOL copybook. The `BigDecimal` class handles packed decimal `COMP-3` fields. The mainframe reader and writer can generate `IOExceptions`.

Listing 11-6 Continuing Imports

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

Step 3: Update EJB with dispatch

In [Listing 11-7](#), the gateway invokes `dispatch` with a byte array of mainframe EBCDIC data. The code converts the mainframe byte array to a data view using a `MainFrameReader`. The `traderData` is the generated data view class.

Listing 11-7 Update EJB with `dispatch`

```
.  
.br/>.br/>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 11-8](#), 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 data view 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 11-8 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

In [Listing 11-9](#), the code converts the data view back into a byte array to be returned to the mainframe using a `MainframeWriter`. The `MainframeWriter` and data view handle conversions. Note that the `dispatch` function takes a byte array and returns a byte array. This means when you set up an initial configuration, you can stub `dispatch` as an echo function.

Listing 11-9 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 11-10](#). The Java gateway must be restarted for new services to take effect.

Listing 11-10 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 basic statelessSession example. The EJB is saved in `/myserver/ejb_basic_statelessSession.jar`. To deploy the EJB, either use the hot deploy feature of the WebLogic Server console, or add an entry to deploy the jar file in the `weblogic.ejb.deploy` property in the `weblogic.properties` file.

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

Warning: Data view 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 11-11](#) 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 11-11 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 11-12](#) 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 11-12 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)
```

11 *Updating an Existing EJB to Service a Mainframe Request*

```
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.

12 Integrating JAM with Crossplex

This section contains a scenario that shows how to develop a single service servlet-based application that invokes a CrossPlex script on the mainframe. 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 [Chapter 4, “Developing Java Applications,”](#) It gives practical examples for using JAM tools, presented as tasks with step-by-step procedures. This scenario depicts the development of a new application. 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.

The following tasks are required to invoke a CrossPlex script on the mainframe:

- [Task 1: Create a CrossPlex Script](#)
- [Task 2: Use eGen COBOL to Create a Base Application](#)
- [Task 3: Create Your Custom Application from the Base Application](#)
- [Task 4: Update the JAM Configurations and Update WebLogic Server Properties](#)
- [Task 5: Deploy Your Application](#)
- [Task 6: Use the Application](#)

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, part number, etc. This inbound data is passed from your application in a container called a record definition.

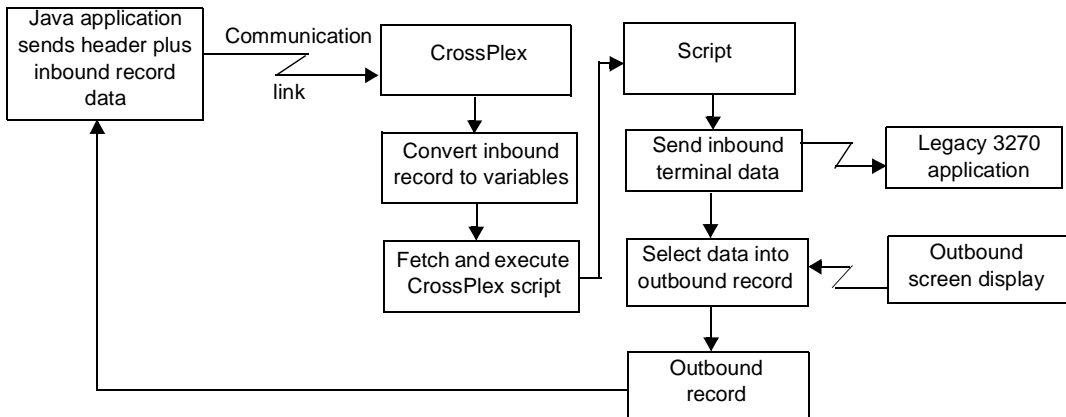
During execution, a script will select and optionally reformat data from the screen displays of the executed 3270 transactions. This selected data will be returned to your application in an outbound record definition.

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 both inbound and outbound data, or two definitions may be used.

Another of the CrossPlex development tools will create 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 12-1](#) illustrates the processing flow from the JAM front end to retrieve data from one or more mainframe transactions.

Figure 12-1 Processing Flow from JAM to Mainframe Transactions

Step 1: Prepare Inbound Record Definition

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, which requires a customer number and company number as input. For this example, the inbound and outbound record definition will be different, though the same record definition can be used for both. [Figure 12-2](#) shows how the inbound record would appear.

Figure 12-2 Inbound record illustration

____ 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	...
***		__0	__0	-	__0	...

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

The inbound 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 Inbound Record Definition

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, would appear as follows:

```
*****
*          INREC - Sample inbound record definition*
*****
01  INREC-START.
    05  INREC-CUSTNOPIC X(007).
    05  INREC-COMPANYPIC 9(003).
```

Step 3: Create an Outbound Record Definition and Copybook

If the outbound data is to use a different record format from the inbound, repeat steps 1 and 2 to prepare the outbound record definition and copybook.

For this example, the outbound record definition and copybook would appear as in [Figure 12-3](#).

Figure 12-3 Outbound Record Definition

```

_____ 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 outbound record definition*
*****
01  OUTREC-START.
    05  OUTREC-CUSTOMERPIC X(007).
    05  OUTREC-NAMEPIC X(025).
    05  OUTREC-ADDRESS1PIC X(025).
    05  OUTREC-ADDRESS2PIC X(025).
    05  OUTREC-CITYPIC X(025).
    05  OUTREC-STATEPIC X(002).
    05  OUTREC-ZIPPIC 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 will navigate 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 inbound terminal data 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 12-1](#).

Listing 12-1 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 outbound
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
NOTFOUNDEnter 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 will execute 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.

Once 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. This is also true 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 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 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 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. This includes 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 12-1](#) lists the four DataView classes created from the three copybooks.

Table 12-1 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 inbound record area (INREC). The data selected in the script will be returned in the outbound record area (OUTREC), which occupies the same memory address as the inbound record, 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. These are:

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

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

Immediately following the header, the inbound and outbound record area must be defined. Since they occupy the same position, the outbound record area should redefine the inbound. The area must appear at field `XP-MESSAGE-AREA` in the header copybook.

[Listing 12-2](#) shows the COBOL version of the header copybook.

Listing 12-2 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.
```

Task 2: Use eGen COBOL to Create a Base Application

```
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-OUTSCREEN               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 12-3](#), the DataViews are generated from the combined copybooks.

Listing 12-3 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 12-4](#) for the CrossPlex operation. This will specify the DataView.

Listing 12-4 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. This is corrected in the custom code written later in the scenario.

Listing 12-5 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 12-6](#), `BaseServlet` is the servlet name to be registered as a URL in the WebLogic Server properties 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 12-6 Add Servlet Name

```
servlet BaseServlet shows page1
```

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

Step 5: Generate the Java Source Code

In [Listing 12-7](#), invoke the eGen COBOL code generator to create the base application that is then compiled. This 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 12-7 Generating the Java Source Code

```
java com.bea.jam.egen.EgenCobol emprec.egen  
ls *.java  
    InrecRecord.java BarHdrRecord.java QuxRecord.java  
    OutrecHdrRecord.java BaseServlet.java
```

Task 3: Create Your Custom Application from the Base Application

The preferred customizing method is to derive a custom class from the generated base 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 12-8](#), `BigDecimal` supports COMP-3 packed data. `HttpSession` is available for saving limited state. `DataView` is the base for all generated data records.

Listing 12-8 Using Imports to Start Creating the Custom Application

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

Step 2: Declare the New Custom Class

[Listing 12-9](#) shows how to extend the generated servlet. This enables regeneration of the base application without destroying customized code. Fields can be added to the copybook without disrupting the customized code.

Listing 12-9 Declaring the New Custom Class

```
public class customServlet
    extends BaseServlet
{
:
:
}
```

Step 3: Add Implementation for doGetSetup

In [Listing 12-10](#), the `doGetSetup ()` function is used to ensure that the user is presented with a form reflecting the `INREC` record.

Listing 12-10 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 12-11](#), the data view 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 12-11 Create Implementation for doPostSetup

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

    // Move the contents, by using a Hashtable as an intermediate holder.
    Hashtable h = new HashtableUnloader().unload(dv);
    new HashtableLoader.load(bhr);

    // Load header fields.
    bhr.getXpCommarea().setXpCommand("EXEC");
    bhr.getXpCommarea().setXpTarget("THISCICS");
    bhr.getXpCommarea().setXpPool("POOLM2");
    bhr.getXpCommarea().setXpFepiConvid(0L);
    bhr.getXpCommarea().setXpMode("CMDR");
    bhr.getXpCommarea().setXpAreaLength((short) 300);
    bhr.getXpCommarea().setXpExecutingScript("MYSCRIPT");
    bhr.getXpCommarea().setXpInboundRecord("INRECRECRD");

    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:

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 an inbound record definition 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 inbound record definition.

MESSAGEAREA	Contains the inbound record when CrossPlex is called and the outbound record 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 12-12](#), 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 12-12 Create Implementation for doPostFinal

```
public DataView doPostFinal(DataView dv, HttpSession s)
{
    OutrecHdrRecord qhr = (OutrecHdrRecord) dv;
    int resp = qhr.getCommarea().getXpCommarea().getXpResponse();
    if (resp != 0 && resp != 12)
        throw new Error("Bad xp-response: " + resp);

    OutrecRecord qr = new OutrecRecord();
```

```
// Move the contents, by using a Hashtable as an intermediate holder.
Hashtable h = new HashtableUnloader().unload(dv);
new HashtableLoader.load(qr);

return qr;
}
```

Task 4: Update the JAM Configurations and Update WebLogic Server Properties

Update the `jcrmgw.cfg` file with the remote service entries shown in [Listing 12-13](#). The Java 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 12-13 Remote Service Entries for Create/Read/Update/Delete

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

Update the `weblogic.properties` file with the entries shown in [Listing 12-14](#).

Listing 12-14 Update WebLogic Server Properties 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. The following are standard WebLogic Server servlet deployment steps:

- If required, compile the *.java files.
- Copy the class files to the WebLogic Server servlet directory.
- If required, add URL registration lines to the weblogic.properties file.
- If required, restart WebLogic Server.
- Test the result with your browser pointed at the registered URL.

Task 6: Use the Application

[Figure 12-4](#) 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 12-4 New Data Entry Servlet Display

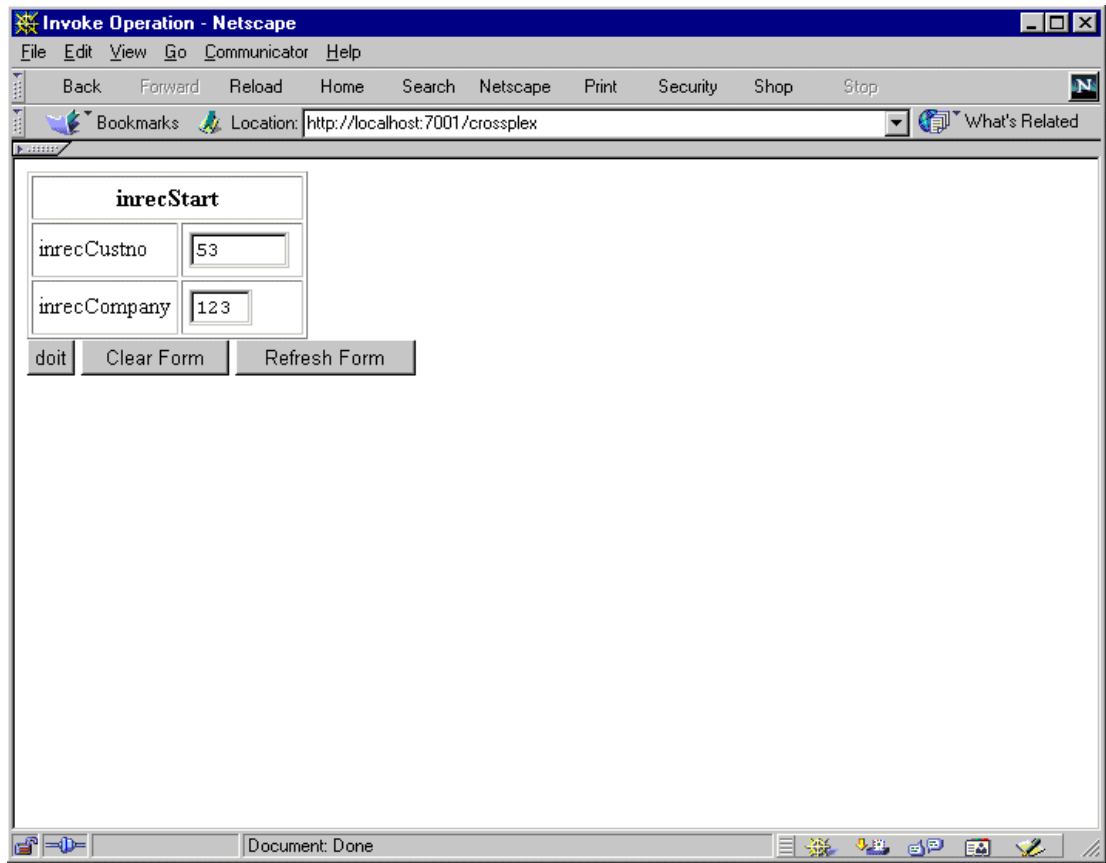


Figure 12-5 is an example of the page used for the front end of the new custom servlet.

Figure 12-5 New Data Entry Servlet Front End Page

The screenshot shows a Netscape browser window titled "Results of Operation - Netscape". The address bar displays "http://localhost:7001/crossplex". The main content area contains a form titled "outrecStart" with the following fields and values:

outrecStart	
outrecCustomer	53
outrecName	Acme
outrecAddress1	123 Main
outrecAddress2	Suite 100
outrecCity	Dallas
outrecState	TX
outrecZip	75123

Below the form are two buttons: "Clear Form" and "Refresh Form". The status bar at the bottom indicates "Document: Done".

A Code Generator Reference Pages

This section contains reference pages for the BEA WebLogic Java Adapter for Mainframe COBOL code generator (eGen COBOL). It describes the rules for writing the script file that controls the code generator.

eGen COBOL

The eGen COBOL tool maps a COBOL copybook into a Java class.

Synopsis

Invoke the tool with the following command:

```
java com.bea.jam.egen.EgenCobol scriptfile
```

where:

`java`

is the name of the Java virtual machine executable in the Java Development Kit (JDK).

`com.bea.jam.egen.EgenCobol`

is the full class name of the eGenCobol tool.

scriptfile

is the script file that controls the eGen COBOL tool. You must write this script file on an application-by-application basis. (See [Listing A-1](#) for an example).

If the JAM installation bin directory has been added to your path, then the eGen COBOL tool may also be invoked with the command:

```
egencobol scriptfile
```

Listing A-1 Example of `scriptfile.egen`

```
### example script
#

view demo.CustomDataView from emprec.cpy

service demoService accepts CustomDataView returns CustomDataView

page demoPage "Demo Page"
{
    view demo.CustomDataView

    buttons
    {
        "Try It" service(demoService) shows demoPage
    }
}

servlet demo.DemoServlet shows demoPage
```

Script Syntax Reserved Words

The reserved words shown in [Table A-1](#) must be used as specified in the “Grammar” section.

Note: A reserved word can be used as an identifier if it is enclosed in either single or double quotation marks (refer to “General Rules”).

Table A-1 Reserved Words

accepts	buttons	class	client	codepage	ejb
from	generate	group	is	method	page
reset	returns	server	service	servlet	shows
support	view	xml			

General Rules

- The '#' character and all following characters on the same line are a comment. Use the '#' character to specify commented text.
- The character sequence " //" and all following characters on the same line are a comment. Use the " //" characters to specify commented text.
- The character sequence " /* " and all following characters until the occurrence of the sequence " */ " are a comment. Use the " /* " characters to specify commented text that extends beyond one line.
- White space (including newlines) is not significant. White space includes newlines, carriage returns, tabs, spaces, etc.
- Any sequence of letters, digits, underscores, or periods is a word.
- Any word that does not match a reserved word is an identifier.
- Any sequence of characters is treated as an identifier if it is enclosed in either single or double quotes. This allows the use of reserved words and sequences that contain spaces.

Grammar

The following grammar is described using a modified Backus-Naur Form (BNF) syntax, such as in many industry-standard software reference guides. It specifies a context-free grammar. Reserved words are shown in bold. Comments are in italics preceded by a dash (*—*).

```
script:
    definition | script definition

fulldefinition:
    generate definition | definition

definition:
    viewdef | servicedef | servletdef | ejbdef | classdef |
    pagedef

viewdef:
    view viewname from copybook | viewdf viewmodifier

viewmodifier:
    codepage codepagename | support xml

servicedef:
    service servicename accepts fullViewname returns
    fullViewname

servletdef:
    servlet classname shows pagename

ejbdef:
    clientejb | serverejb

clientejb:
    client ejb classname ejbregistration { clientmethods }

serverejb:
    server ejb classname ejbregistration { servermethods }

classdef:
    client class classname { clientmethods }

pagedef:
    page pagename title { view viewname buttons { buttonlist } }

buttonlist:
    buttondef | buttonlist buttondef

buttondef:
    servicebutton | ejbbutton
```

```

clientmethods:
    clientmethoddef | clientmethods clientmethoddef
clientmethoddef:
    method methodname is servicename
servermethods:
    servermehtoddef | servermethods servermethodddef
servermethoddef:
    method methodname (fullviewname) returns fullviewname
servicebutton:
    buttonname service ( servicename ) shows pagename
ejbbutton:
    buttonname ejbmethod ( fullViewname ) returns fullViewname
viewname:
    classname
fullViewname:
    viewname | viewname [ codepagename ]
copybook:
    identifier
    —An identifier that names a file containing a COBOL data definition.
servicename:
    identifier
    —An identifier that matches a resource definition in your jcrmgw.cfg file
pagename:
    identifier
    —An identifier that names a page definition.
codepagename:
    identifier
    —The name of a codepage to be used for character translation to/from
    mainframe data formats. This must be a codepage supported by the JDK
    being used.
methodname:
    identifier
    —The name to be given to a generated Java method.
classname:
    identifier
    —An identifier that names a Java class, including any package name.

```

`ejbregistration:`
 `identifier`
 —The name that will be used to register the home interface for an EJB.

`title:`
 `identifier`
 —The title to be placed into the HTML generated for a page.

`buttonname:`
 `identifier`
 —A button name that will be used in the HTML generated for a page.

`ejbmethod:`
 `identifier`
 —An EJB classname and method specification that should look like this:
 `package.ejbclass.method`
 or
 `ejbclass.method`

Results of Running the Code Generator

- Each view definition (described in the “[Programming Reference](#)” section of this guide) causes the specified COBOL copybook to be parsed and a Java source file for the specified `DataView` class to be generated in the current directory.

If XML support was requested, then the following files are also produced:

- `viewnameHelper.Java` - XML Helper class for use with WebLogic Process Integrator
- `viewname.dtd` - DTD file
- `viewname.xsd` - XML Schema file
- Each servlet definition causes a Java source file to be generated in the current directory for the specified class.
- Each client class definition causes a Java source file to be generated in the current directory for the specified class.
- Each EJB definition causes the generation of three Java source files and a deployment descriptor text file into the current directory. The names of the generated files are listed in [Table A-2](#).

Table A-2 Generated Files for EJB Definitions

Name of File	Purpose
<i>classnameHome.java</i>	EJB Home Interface
<i>classnameBean.java</i>	EJB Implementation class
<i>classname.java</i>	EJB Remote Interface
<i>classname-jar.xml</i>	EJB Deployment descriptor
<i>wl-classname-jar.xml</i>	WebLogic Deployment Info

B Configuration Checker Utility

The Java Communications Resource Manager Gateway (JCRMGW) is a Java application that manages sessions providing access into and out of the Java environment. It is configured using a text file named `jcrmgw.cfg` that resides in the WebLogic server directory. This configuration file is processed every time the gateway starts. Requests coming from the mainframe are mapped to an EJB that services the request while requests going to the mainframe are mapped to a mainframe program which can be executed using a CICS DPL. The JCRMGW also supports IMS operations.

When making changes to this file, you can verify the contents by invoking the gateway's configuration processor directly from the command line. This verification process allows you to discover and correct any errors prior to starting the gateway.

This topic consists of the following sub-topics:

- [bea.sna.jcrmgw.jcrmConfigurator](#)

This topic describes the configuration file checker utility.

- [GWBOOT](#)

This topic describes the `gwboot` startup class and its options.

bea.sna.jcrmgw.jcrmConfigurator

Java Communications Resource Manager Gateway configuration (`jcrmgw.cfg`) checker class.

Synopsis

`bea.sna.jcrmgw.jcrmConfigurator`

Description

The `bea.sna.jcrmgw.jcrmConfigurator` is used to check the `jcrmgw.cfg` file prior to starting the `jcrmgw`. It is recommended to place it into a script file and run it with standard output redirected to a file. The resulting output will be either diagnostic messages indicating syntax errors in the configuration file or a formatted listing of the definitions as they will be used by the gateway.

There are no options and the only file which can be processed is the `./jcrmgw.cfg` file in the current directory.

GWBOOT

JCRMGW bootstrap command. This `gwboot` startup class launches the gateway when the server is started.

Synopsis

```
bea.sna.jcrgmw.gwboot [[-r] [-t] [-u] <userid> ]
```

Description

`gwboot` is used in the `weblogic.properties` file to start an instance of the `jcrgmw` and optionally spawn a CRM to listen on the address specified in the `jcrgmw.cfg` file if required.

The following options can be specified:

`-r`

specifies a remote SNACRM and suppresses the spawning of a CRM process. The CRM is assumed to already be running and listening at the address specified in the SNACRMADDR entry of the `jcrgmw.cfg` file.

This would be required when the CRM is running on a machine other than the gateway, or if the CRM was started in its own window for purposes of testing a configuration.

`-t`

specifies tracing CRM. Turns on level 3 CRM tracing. This can only be used if the gateway is spawning a new CRM. If the CRM is running remotely and tracing is required, use the `-t` option on the command line which started the CRM.

`-u<userid>`

specifies `<userid>`. This is the default userid to be used by the gateway for all APPC requests. This is useful for IDENTITY type security when the gateway clients do not set a userid.

C Error and Informational Messages

The following table contains a description of error, informational, and warning messages that can be encountered while using the JAM software.

100	warning: 66 level (RENAMES) is not supported
DESCRIPTION	This language feature is not supported.
ACTION	No action is necessary.
101	warning: 88 level (condition name) is not supported
DESCRIPTION	This language feature is not supported.
ACTION	No action is necessary.
102	warning: Binary bitfield datatype is not supported
DESCRIPTION	This language feature is not supported.
ACTION	No action is necessary.
103	warning: COMP-5 datatype is not supported
DESCRIPTION	This language feature is not supported.
ACTION	No action is necessary, but it is recommended that the source file be corrected.

104	warning: COMP-X datatype is not supported
DESCRIPTION	This language feature is not supported.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
105	warning: Extraneous ' ' ignored
DESCRIPTION	A extra delimiter was encountered, and is ignored.
ACTION	No action is necessary.
106	warning: Extraneous OCCURS TO clause, ignored
DESCRIPTION	This clause is not necessary, and is ignored.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
107	warning: INDEXED BY clause ignored
DESCRIPTION	This clause is not necessary, and is ignored.
ACTION	No action is necessary.
108	warning: Identifier is not unique: {name}
DESCRIPTION	The data item name is not unique, which might cause ambiguity.
ACTION	No action is necessary.
109	warning: KEY IS clause ignored
DESCRIPTION	This clause is not necessary, and is ignored.
ACTION	No action is necessary.
110	warning: Level number {num} is out of sequence, treating like {num}
DESCRIPTION	The level number of a data item definition does not match previous level numbers, so a default value is assumed.
ACTION	No action is necessary, but it is recommended that the source file be corrected.

111	warning: OCCURS lower bound exceeds upper bound ({occurMin} > {occurMax})
DESCRIPTION	The OCCURS ranges are out of order.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
112	warning: PICTURE ignored for COMP-1/COMP-2 datatype
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary.
113	warning: PICTURE ignored for INDEX datatype
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
114	warning: PICTURE ignored for POINTER datatype
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
115	warning: PICTURE ignored for binary bitfield datatype
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
118	warning: Token begins with an unrecognizable character ({char})
DESCRIPTION	An unrecognizable character was encountered in the source file.
ACTION	No action is necessary, but it is recommended that the source file be corrected.

119	warning: USAGE ignored for 88-level datatype
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
120	warning: Data item follows a 66-level item
DESCRIPTION	All 66-level items must be the last items within a given group.
ACTION	Correct the source file.
121	warning: JUSTIFY clause ignored for non-alphanumeric item
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
122	warning: PICTURE clause ignored for RENAMES datatype
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
123	warning: PICTURE clause required for NO USAGE datatype
DESCRIPTION	The required clause is missing.
ACTION	Correct the source file.
124	warning: SIGN clause ignored
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.

125	warning: Terminating “.” appears to be missing
DESCRIPTION	Data Record definitions must be terminated with a !!
ACTION	No action is necessary, but it is recommended that the source file be corrected.
126	warning: USAGE clause ignored for RENAMEs item
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	No action is necessary, but it is recommended that the source file be corrected.
127	warning: OCCURS lower bound assumed to be 1
DESCRIPTION	The lower bound in the OCCURS clause of the DEPENDS ON clause is missing and assumed to be 1.
ACTION	Items with a DEPENDS ON clause require an OCCURS lower bound.
128	warning: OCCURS lower and upper bounds should be different
DESCRIPTION	The upper and lower bound in the OCCURS clause of the DEPENDS ON clause are the same.
ACTION	Items with a DEPENDS ON clause should have different upper and lower bounds.
200	error: BLANK WHEN ZERO clause ignored for non-zoned item
DESCRIPTION	The clause is not meaningful for the data item definition.
ACTION	Correct the source file.
201	error: Bad data item clause
DESCRIPTION	A syntax error or semantic disagreement was encountered while parsing the data item definition.
ACTION	Correct the source file.
202	error: Cannot REDEFINE the item: {name}

	DESCRIPTION	The name specified is not a valid REDEFINES target.
	ACTION	Correct the source file.
203	error: Character literal is missing its closing quote	
	DESCRIPTION	Quoted literals require a closing quote mark.
	ACTION	Correct the source file.
204	error: Character literal is too long, truncated ({num})	
	DESCRIPTION	Character literals are truncated beyond a fixed upper limit.
	ACTION	Correct the source file.
205	error: DEPENDING ON clause requires OCCURS TO upper bound	
	DESCRIPTION	The required clause is missing.
	ACTION	Correct the source file.
206	error: DEPENDING ON item is not an integer: {name}	
	DESCRIPTION	The DEPENDING ON data item is not a numeric integer type.
	ACTION	Correct the source file.
207	error: DEPENDING ON clause requires an OCCURS clause	
	DESCRIPTION	The required clause is missing.
	ACTION	Correct the source file.
208	error: Expected an ASCENDING/DESCENDING KEY IS clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
209	error: Expected BLANK	
	DESCRIPTION	A syntax error occurred while parsing the source file.

	ACTION	Correct the source file.
210	error: Expected a <code>DEPENDING ON</code> clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
211	error: Expected a <code>DEPENDING ON</code> qualified identifier	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
212	error: Expected <code>EXTERNAL</code>	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
213	error: Expected <code>EXTERNAL/GLOBAL</code>	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
214	error: Expected <code>GLOBAL</code>	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
215	error: Expected an <code>INDEXED BY</code> clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
216	error: Expected an <code>INDEXED BY</code> qualified identifier	
	DESCRIPTION	A syntax error occurred while parsing the source file.

	ACTION	Correct the source file.
217	error: Expected a JUSTIFIED clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
218	error: Expected a KEY IS qualified identifier	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
219	error: Expected LEADING/TRAILING, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
220	error: Expected an OCCURS clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
221	error: Expected OCCURS lower bound, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
222	error: Expected OCCURS upper bound, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
223	error: Expected a PICTURE clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.

	ACTION	Correct the source file.
224	error: Expected a PICTURE specification, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
225	error: Expected a REDEFINES clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
226	error: Expected a REDEFINES identifier, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
227	error: Expected a RENAMES THRU identifier, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
228	error: Expected a RENAMES clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
229	error: Expected a RENAMES qualified identifier, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
230	error: Expected a SIGN clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.

	ACTION	Correct the source file.
231	error: Expected a SYNC clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
232	error: Expected a VALUE clause	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
233	error: Expected ZERO	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
234	error: Expected a ')' following a bitfield size, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
235	error: Expected a USAGE data type, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
236	error: Expected a bitfield size, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
237	error: Expected a data clause, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.

	ACTION	Correct the source file.
238	error: Expected a level number, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
239	error: Expected an identifier or FILLER, found '{text}'	
	DESCRIPTION	A syntax error occurred while parsing the source file.
	ACTION	Correct the source file.
240	error: Extraneous BLANK WHEN ZERO clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
241	error: Extraneous DEPENDING ON clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
242	error: Extraneous EXTERNAL clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
243	error: Extraneous GLOBAL clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
244	error: Extraneous INDEXED BY clause	
	DESCRIPTION	The data item definition can only have one such clause.

	ACTION	Correct the source file.
245	error: Extraneous JUSTIFY clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
246	error: Extraneous KEY IS clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
247	error: Extraneous OCCURS clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
248	error: Extraneous PICTURE clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
249	error: Extraneous REDEFINES clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
250	error: Extraneous RENAMES clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
251	error: Extraneous SIGN clause	
	DESCRIPTION	The data item definition can only have one such clause.

	ACTION	Correct the source file.
252	error: Extraneous SYNC clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
253	error: Extraneous USAGE clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
254	error: Extraneous VALUES clause	
	DESCRIPTION	The data item definition can only have one such clause.
	ACTION	Correct the source file.
255	error: Hex string literal must have an even number of digits	
	DESCRIPTION	Hexadecimal character literals must be composed of an even number of digits.
	ACTION	Correct the source file.
256	error: INDEXED BY clause requires an OCCURS clause	
	DESCRIPTION	The required clause is missing.
	ACTION	Correct the source file.
257	error: Improper bitfield size ({len})	
	DESCRIPTION	A improper length was specified.
	ACTION	Correct the source file.
258	error: Improper level-number for REDEFINES item ({levelNo})	
	DESCRIPTION	The level numbers of redefined data items must match.
	ACTION	Correct the source file.

260	error: KEY IS clause requires an OCCURS clause
DESCRIPTION	The required clause is missing.
ACTION	Correct the source file.
261	error: Level number {n} is out of sequence, treating like {n}
DESCRIPTION	The level number of a data item definition does not match previous level numbers, so a default value is assumed.
ACTION	Correct the source file.
262	error: Malformed DEPENDING ON clause
DESCRIPTION	A syntax error occurred while parsing the source file.
ACTION	Correct the source file.
263	error: Malformed INDEXED BY clause
DESCRIPTION	A syntax error occurred while parsing the source file.
ACTION	Correct the source file.
264	error: Malformed KEY IS clause
DESCRIPTION	A syntax error occurred while parsing the source file.
ACTION	Correct the source file.
265	error: Malformed USAGE clause
DESCRIPTION	A syntax error occurred while parsing the source file.
ACTION	Correct the source file.
267	error: Malformed VALUE clause
DESCRIPTION	A syntax error occurred while parsing the source file.
ACTION	Correct the source file.

268	error: Malformed data definition ignored for: {name}
DESCRIPTION	A syntax error occurred while parsing the source file.
ACTION	Correct the source file.
269	error: Malformed data definition, ignored
DESCRIPTION	A syntax error occurred while parsing the source file.
ACTION	Correct the source file.
270	error: Malformed picture specification: '{pic}'
DESCRIPTION	The picture clause contains invalid characters.
ACTION	Correct the source file.
271	error: Missing PICTURE clause
DESCRIPTION	The data item definition requires such a clause.
ACTION	Correct the source file.
272	error: Missing USAGE and PICTURE clauses
DESCRIPTION	The data item definition requires such a clause.
ACTION	Correct the source file.
273	error: Missing VALUE literal constant, found '{text}'
DESCRIPTION	The clause contains a syntax error.
ACTION	Correct the source file.
274	error: Missing identifier following IN/OF, found '{text}'
DESCRIPTION	The clause contains a syntax error.
ACTION	Correct the source file.
275	error: Missing literal constant after THRU, found '{text}'
DESCRIPTION	The clause contains a syntax error.

	ACTION	Correct the source file.
276	error: Nonexistent or nonunique DEPENDING ON identifier: {name}	
	DESCRIPTION	The name specified is ambiguous.
	ACTION	Correct the source file.
277	error: OCCURS count must be greater than zero ({occurMax})	
	DESCRIPTION	Arrays must have at least one element.
	ACTION	Correct the source file.
278	error: REDEFINES identifier cannot be qualified	
	DESCRIPTION	The clause contains a syntax error.
	ACTION	Correct the source file.
279	error: REDEFINES item must have the same level number ({levelNo})	
	DESCRIPTION	The level numbers of redefined data items must match.
	ACTION	Correct the source file.
281	error: Recovering, skipping to next '.'	
	DESCRIPTION	A syntax error was encountered, so the rest of the definition is ignored.
	ACTION	Correct the source file.
282	error: String literal is empty	
	DESCRIPTION	A quoted literal must contain at least one character.
	ACTION	Correct the source file.
283	error: USAGE and PICTURE clauses disagree	
	DESCRIPTION	The clauses specify contradictory types or lengths.
	ACTION	Correct the source file.
284	error: Word is too long, truncated ({num})	

	DESCRIPTION	Token words cannot be longer than a certain fixed length.
	ACTION	Correct the source file.
285	error: PICTURE and SIGN clauses disagree	
	DESCRIPTION	The clauses specify contradictory types or lengths.
	ACTION	Correct the source file.
300	Error: An I/O error occurred while generating [{name}]: {error}	
	DESCRIPTION	Could not write to the output file.
	ACTION	Check the permissions of the output file.
301	Error: An I/O error occurred while generating [{name}]: {error}	
	DESCRIPTION	Could not write to the output file.
	ACTION	Check the permissions of the output file.
302	Error: An I/O error occurred while generating [{name}]: {error}	
	DESCRIPTION	Could not write to the output file.
	ACTION	Check the permissions of the output file.
303	Error: An I/O error occurred while generating [{view}]: {error}	
	DESCRIPTION	Could not write to the output file.
	ACTION	Check the permissions of the output file.
304	Error: An I/O error occurred while reading the script: {file}	
	DESCRIPTION	The file could not be read.
	ACTION	Check the permissions of the input file.
305	Error: EJB specification must contain both a class name and a method name	
	DESCRIPTION	Proper code cannot be generated without the missing items.
	ACTION	Provide the missing items.

306	Error: EJB {bean} is not defined.
DESCRIPTION	A nonexistent EJB bean name was referenced.
ACTION	Specify a different file name.
307	Error: Parse failed on [{file}].
DESCRIPTION	A syntax error was encountered while parsing the script file.
ACTION	Correct the script.
308	Error: The copybook [{file}] was not found.
DESCRIPTION	A nonexistent COBOL source file name was specified.
ACTION	Correct the misspelling or provide the missing source file.
309	Error: The script file [{file}] was not found.
DESCRIPTION	A nonexistent file name was specified.
ACTION	Specify a different file name.
310	Error: excess method {name} is ignored.
DESCRIPTION	An extraneous method definition was specified.
ACTION	Remove the duplicate definition.
311	Error: expecting {token}.
DESCRIPTION	A syntax error occurred while parsing the input file.
ACTION	Correct the input file.
312	Error: method {name} is not defined in EJB {bean}.
DESCRIPTION	A nonexistent method was referenced.
ACTION	Correct the input file.
313	Error: service {name} is not defined.
DESCRIPTION	A nonexistent service name was referenced.

	ACTION	Correct the input file.
314	Error: service {service} is not defined.	
	DESCRIPTION	The service name referenced was not defined.
	ACTION	Provide the missing service name definition or correct the misspelling.
315	Error: servlet {name} refers to an unknown page ({page}).	
	DESCRIPTION	A nonexistent page name was referenced.
	ACTION	Correct the input file.

D Java Docs

The BEA WebLogic Java Adapter for Mainframe (JAM) product comes with HTML pages that document the JAM Java classes. These also are referred to as “javadoc” files. They are located in the `jamdoc.jar` file, found in the JAM installation directory.

Issue the following command to extract the javadoc HTML files from the jar file:

```
jar -xvf jamdoc.jar
```

where:

`jar`

is the Java archive command.

`-xvf`

is the extract file(s) parameter.

`v`

is the verbose parameter to list the files.

`f`

is the option which designates the next parameter as the jar file name.

`jamdoc.jar`

is the name of the JAM javadoc file.

This command extracts all of the files contained in the jar file into the current directory. The HTML documentation files are placed in a newly created subdirectory named `classdocs` in the current directory.

To view an HTML documentation file, open your web browser and specify the file name of the javadoc you want to view, taken from the `classdocs` directory. Any of the following files are good for getting started:

- `classdocs/AllNames.html`
- `classdocs/packages.html`
- `classdocs/tree.html`

Glossary

A

Application Programming Interface (API)

1) The verbs and environment that exist at the application level to support a particular system software product. 2) A set of code that enables a developer to initiate and complete client/server requests within an application. 3) A set of calling conventions that define how to invoke a service. A set of well-defined programming interfaces (entry points, calling parameters, and return values) by which one software program utilizes the services of another

Application Program-to-Program Communication (APPC)

An interface to LU6.2 services; provides a set of primitives to conduct conversations in LU6.2 sessions.

B

buffer

see "typed data buffer."

C

CICS

"Customer Information Control System." A program execution monitor system resembling an operating system that controls and manages the execution of multiple transactional programs. CICS programs are arranged within "regions." COBOL is the primary programming language used for CICS applications.

COBOL

"Common Business Oriented Language." A semi-structured programming language designed by the U.S. DoD in 1959, designed to be used for most business applications. It is the most common language used for most mainframe applications in general.

COMMAREA

A CICS data buffer area that is sent to application programs, typically as a COBOL linkage section variable, providing an area for sending data to, and returning data from, application programs. This is part of the data sent to a program invoked via DPL.

copybook

A COBOL source file that is included by one or more source programs and typically contains a declaration for one or more record (group) data items.

D

DataView

The representation of a data record, consisting of information about all of the data fields within the record as well as their types, lengths, and offsets.

DPL

"Distributed Program Link." A remote invocation of a CICS program from another CICS region or SNA domain.

Domain

A *domain* can be another BEA WebLogic Server application that is independently administered, an application that is under the control of another transaction processing system, or an application in a remote CICS/ESA region. Domains can be local or remote.

E

EJB

"Enterprise Java Beans." The object-oriented business component model for transactional-based programs, defining the operation and protocols between Java client and server programs.

I

Information Management System (IMS)

A database manager used by CICS/ESA to allow access to data. IMS provides for the arrangement of data in an hierarchical structure and a common access approach in application programs that manipulate IMS databases.

J

J2EE

"Java to Enterprise Edition." A collection of Java standards (EJB, JNI, and so forth) that together define an enterprise application programming environment.

Java

An object-oriented programming language. A descendant from C, it was invented by James Gosling of Sun Microsystems.

L

Local Domain

A *Local Domain* is a part of an application (set or subset of services) that is available to other domains. A Local Domain is always represented by a Domain Gateway Group, and the terms are used interchangeably.

Local Service

A *Local Service* is a service of a local domain that is made available to remote domains through a Domain Gateway Group.

Logical Unit (LU)

In SNA, a port through which a user gains access to the services of a network. Also, see System Network Architecture (SNA).

LU6.2

LU6.2 is a particular SNA logical unit that identifies a specific set of services for program to program communication. Services include syncpoint, mapping of buffers into records, message confirmation, and security.

P

Partitioned Data Set (PDS).

A CICS/ESA data set in direct access storage that is divided into partitions called members. A member can contain a program or data. Program libraries are held in partitioned data sets.

R

re-entrant

The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

Remote Domain

A *Remote Domain* is a part of an application accessed through a Domain Gateway Group. The remote domain may be another BEA WebLogic Server application, an application running under another TP system, or a BEA WebLogic Java Adapter for Mainframe application.

Remote Service

A *Remote Service* is a service of a remote domain that is made available to the local application through a Domain Gateway Group.

S

Session

When two LU bind with each other, that is, when they have successfully negotiated how they will communicate, they are said to be in *session*. SNA has fixed limits on the number of sessions configured for an LU type.

Stack

Platform vendor-supplied software that provides connectivity to an SNA network.

SNA Communication Resource Manager (SNACRM)

A process that directly communicates with the PU2.1 server.

System Network Architecture (SNA)

A seven-layer networking protocol. Each layer of the protocol has a set of associated data communication services. The services of the uppermost layer are embodied in a Logical Unit (LU). Each LU type defined in SNA has its own specific set of services available to an end user for communicating. The end user may be a terminal device, or an application program. The SNA structure enables the end user to operate independently, unaffected by the specific facilities used for information exchange.

T

Transaction

- 1) A complete unit of work that transforms a database from one consistent state to another. In DTP, a transaction can include multiple units of work performed on one or more systems.
- 2) A logical construct through which applications perform work on shared resources (e.g., databases). The work done on behalf of the transaction conforms to the four ACID Properties: atomicity, consistency, isolation, and durability.

typed data buffer

A block of data, arranged as a record composed of one or more data fields. Each field has a type and a length, and optionally a name. Buffer types are specified by their name, which typically corresponds to the copybook name or class name that defines the record structure.

V

Virtual Telecommunications Access Method (VTAM)

A set of programs that control communication across a network between terminals and application programs.



Index

A

- accessors 7-4
- Adobe Acrobat Reader xvi
- alphanumeric field
 - rules for mapping 7-3
- APPC 1-11
 - sign-on security style in JAM
 - applications 6-1
- Application View EJB 8-4
- array field
 - rules for mapping 7-5

B

- BigDecimal
 - rules for mapping to 7-3
- BLANK WHEN ZERO field
 - rules for mapping 7-3
- Browser session flowchart 4-22
- Business Operations
 - setting up Jam Components as 8-8

C

- CICS
 - COMMAREA specified by COBOL
 - copybook 4-2
 - Link command 3-1
- CICS application 1-11
- CICS DPL 3-11
- COBOL copybook 1-13, 4-1

- creating new 4-5
- LINKAGE SECTION 4-5
- processing by EngenCobol code
 - generator A-6
- rules for mapping into a Java class 7-1
- rules for mapping REDEFINES 7-5
- using existing 4-5

- COBOL data types
 - syntax features and data types supported
 - by EngenCobol tool 7-6
- configuration
 - defining new gateways 3-2
- context-free grammar
 - rules for EngenCobol code generator
 - script A-4
- conversations between applications xii

D

- Distributed Program Link (DPL) 1-11

E

- edited numeric field
 - rules for mapping 7-3
- EGEN translator
 - generation models 1-13
- EngenCobol code generator
 - rules for generating code 7-1
 - rules for writing script file A-1
- EJB
 - Home Interface class generated by

- EgenCobol code generator A-7
- hot-deployed 5-3
- Implementation class generated by
 - EgenCobol code generator A-7
- Remote Interface class generated by
 - EgenCobol code generator A-7
- EJB application
 - customizing 4-47
 - customizing Java classes in 4-34
 - deploying 5-1
 - generating as a remote server 4-41
 - producing Java classes for 4-26
 - sample script for defining 4-27
 - sample script process command 4-29
 - Stateless Session EJB 4-36
- elementary field
 - rules for mapping 7-4

F

- field name
 - rules for mapping into Java name 7-2
- field type
 - rules for mapping into Java type 7-2
- FUNCTION
 - as optional keyword for remote services 3-13

G

- group field
 - nested, rules for mapping 7-3
 - rules for mapping 7-3
- gwboot startup class B-1

H

- HTML
 - Java Docs E-1
 - Web User Interface for this guide xv

I

- IMS 1-14
- IMS application 1-11
- INDEX field
 - rules for mapping 7-3

J

- JAM
 - XML Capabilities 2-3
- JAM Java classes
 - documented in Java Docs E-1
- jar file
 - extracting Java Docs from E-1
 - jam_11.jar file on product CDROM 7-1
- Java
 - request/response 1-11
- Java application
 - customizing servlet-only application 4-17
 - customizing simple stand-alone application 4-55
 - generating a simple stand-alone application 4-52
 - generating servlet-only application 4-11
 - generating the base Java application 4-1
 - models 4-11
 - sample of generated source file 4-17
 - typed data buffer 4-2
- Java clients
 - send/receive data buffers with
 - COBOL/CICS DPL programs 4-2
- Java data types
 - converting to COBOL data types 4-5
- Java Development Kit (JDK) A-1
- JCRMGW 1-15
 - gwboot command B-3
 - sample configuration file 3-14
 - setting address of associated SNACRM 3-4

- setting tracing for B-3
- using GROUP parameter to correlate
 - with a SNACRM 3-5
- jcrmgw.cfg file 3-2
 - verifying with configuration checker
 - utility B-2
- JUSTIFIED RIGHT field
 - rules for mapping 7-3

L

- Local LU
 - SNA stack parameter for 1-15

M

- Mainframe Application Alias 3-3
- Mainframe Applications
 - mapping program names to EJB servers 3-10
 - mapping program names to method names 3-12
- MAX sessions
 - SNA stack parameter 1-15
- Minimum Contention Winners
 - SNA stack parameter 1-15
- Mode name
 - SNA stack parameter 1-15

N

- numeric field
 - rules for mapping 7-3

O

- OS390/CICS programs xi

P

- password
 - security settings for 6-1

R

- RDOM
 - as keyword for remote services 3-13
- REDEFINES clause
 - rules for mapping 7-5
- Remote LU
 - setting alias for SNACRM partner application 3-9
 - SNA stack parameter 1-15
- remote names xii
- RNAME
 - as keyword for remote services 3-13

S

- Script process command 4-16
- Security
 - RACF profile for applications 6-1
 - settings 3-9
- servlet
 - accessing with URL 5-2
 - deploying 5-1
- SIGN IS LEADING field
 - rules for mapping 7-3
- SIGN IS TRAILING field
 - rules for mapping 7-3
- SNA
 - maximum number of sessions per link 3-8
 - setting minimum number of sessions as contention winners 3-8
- SNA stack 1-15
- SNACRM
 - identifying partner mainframe application regions 3-7
 - setting address in associated JCRMGW 3-4
 - specifying Local LU for 3-6
 - specifying stack for 3-6
 - using GROUP parameter to correlate with a JCRMGW 3-5

stack configuration xii

T

Technical support xviii

TRANTIME

- as optional keyword for remote services 3-14

TUXEDO application administrator xii

U

userid

- security settings for 6-1
- setting in JCRMGW B-3

V

VTAM

- mode table 3-9

W

Web browser

- viewing Java Docs in E-1

Web User Interface (WUI)

- HTML interface for this guide xv

WebLogic Process Integrato

- integrating with JAM 8-1

WebLogic Process Integrator

- architecture 8-2
- example of integration with JAM 8-8
- overview 8-2

WebLogic Server

- as Java server 4-23
- deploying applications to WLS system 5-1

WebLogic Server (WLS) 1-11

WEBLOGICCLASSPATH 4-61

weblogics.properties file

- modifying to deploy servlet 5-1

Workflow

setting up 8-13

Workflow Development 8-5

X

XML

- DTD 2-2

- Schema 2-2

- Understanding how JAM uses XML 2-1

- varieties 2-1

- What XML is 2-1