BEA WebLogic Java Adapter for Mainframe

Programming Guide

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

Copyright

Restricted Rights Legend

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

Trademarks or Service Marks

BEA WebLogic Java Adapter for Mainframe Programming Guide

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

Contents

About This Document

1. Generating a Java Application with the eGen COBOL Code Generator

2. Generating a Servlet-Only Application

3. Generating a Client Enterprise Java Bean-based Application

4. Generating a Server Enterprise Java Bean-based Application

About This Document

The BEA WebLogic Java Adapter for Mainframe product (hereafter referred to as JAM) is a gateway connectivity application that enables client/server transactions between Java applications and OS/390 or IMS programs. In addition to the runtime environment that provides the gateway connectivity, JAM also provides tools for developing cooperative Java applications.

This document provides the following topics on installing JAM software:

- "Generating a Java Application with the eGen COBOL Code Generator" describes the tasks in general required to generate a Java application that can be used to make mainframe application requests.
- "Generating a Servlet-Only Application" describes the tasks required to generate a servlet that can be used to make mainframe application requests.
- "Generating a Client Enterprise Java Bean-based Application" describes the tasks required to generate a Client EJB application that can be used to make mainframe application requests.
- "Generating a Server Enterprise Java Bean-based Application" describes the tasks required to generate a Server EJB application that can be used to accept mainframe application requests.
- "Generating a Stand-alone Client Application" describes the tasks in general required to generate a Java stand-alone client application that can be used to make mainframe application requests.

What You Need to Know

This document is intended for application programmers who will develop applications using the development tools included with BEA WebLogic Java Adapter for Mainframe.

e-docs Web Site

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

How to Print the Document

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

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

Related Information

The following BEA publications are available for JAM 4.2:

- *BEA WebLogic Java Adapter for Mainframe Release Notes*

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

Contact Us

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

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

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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

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

Convention	Item

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

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

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

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

CHAPTER

1 Generating a Java Application with the eGen COBOL Code Generator

The BEA WebLogic Java Adapter for Mainframe (JAM) consists of two components:

- JAM gateway
- Communication Resource Manager (CRM)

Using JAM, BEA WebLogic Server users can make requests for mainframe services and receive responses to those requests. Also, mainframe users can make requests from Java applications (EJBs) running in WebLogic Server and receive responses to those requests.

This Programming Guide provides you with instructions on generating Java applications that make requests for mainframe services and accept the responses returning from the mainframe. One of the generated application types, the Server EJB, accepts requests from mainframe applications, then responds to those requests. Refer to the action list in the “Action List” section.

Action List

As you generate a Java application with the eGen COBOL Code Generator (also called the eGen utility), see the following action list and refer to the appropriate information sources.

	Your action...	Refer to...
1	Complete all prerequisite tasks.	“Prerequisites”
2	Choose an eGen application model.	“Understanding JAM” and “Choosing an eGen Java Application Model”
3	Gather necessary mainframe applications information.	“Gathering Mainframe Applications Information”
4	Write an eGen COBOL script	“Writing an eGen COBOL Script”
5	Process eGen Scripts with the eGen utility	“Processing eGen Scripts with the eGen Utility”
6	Compile the Java application code.	“Special Considerations for Compiling the Java Code”
7	Deploy the generated applications.	“Deploying Applications”
8	Debug the generated applications.	“Using Client Diagnostic Features with WebLogic Server 6.0”
9	Proceed to the next set of instructions.	“What Do I Do Next?”

Prerequisites

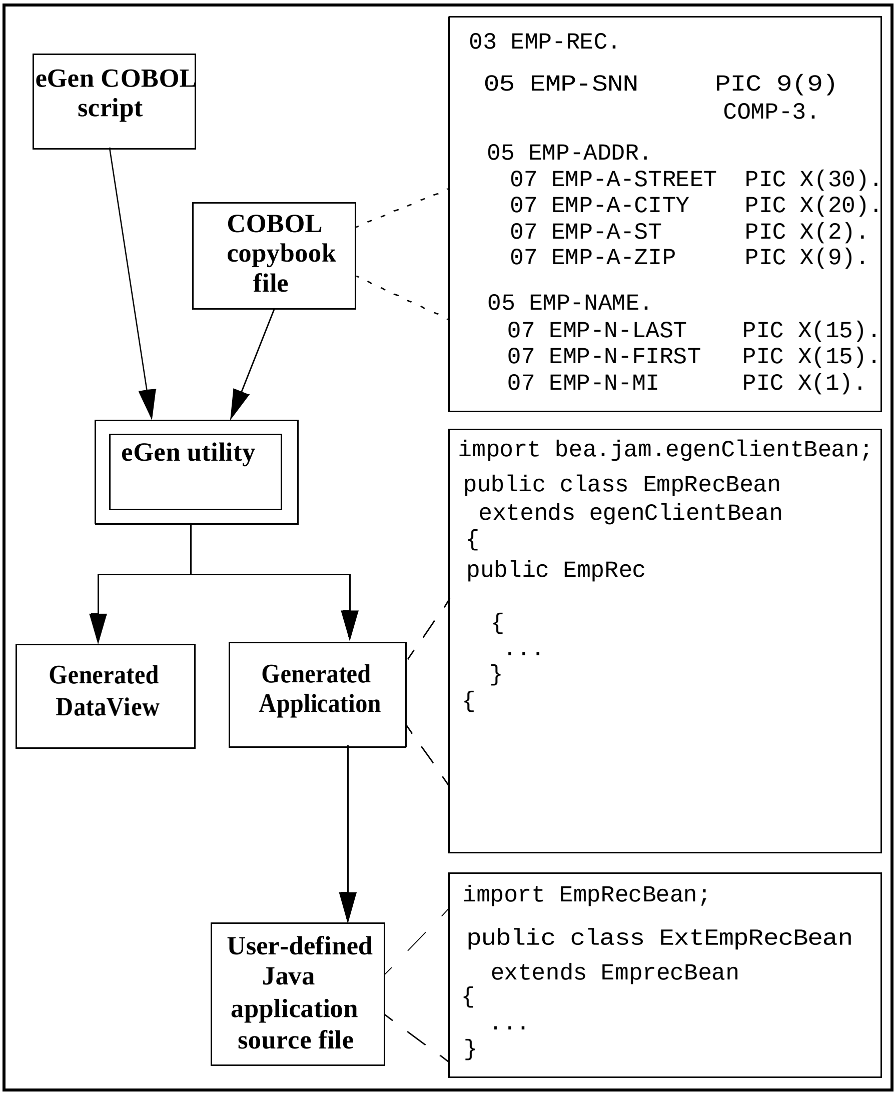
Before you start programming, you should complete the following tasks:

	Your action...	**Refer to...**
1	Install your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Configure your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*

Understanding JAM

Figure 1-1 illustrates how the eGen utility works.

Figure 1-1 Understanding the eGen Utility



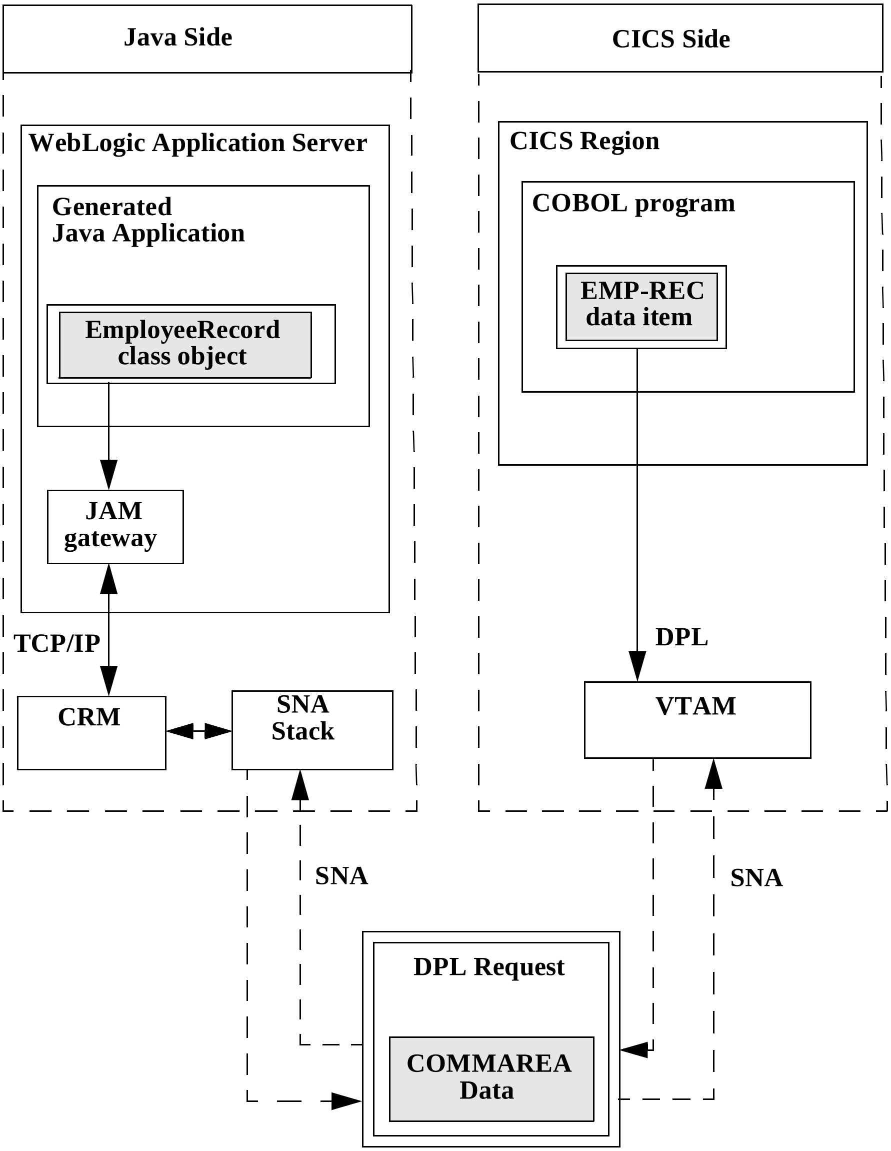
```
03 EMP-REC.

 05 EMP-SNN      PIC 9(9)
                  COMP-3.

 05 EMP-ADDR.
   07 EMP-A-STREET  PIC X(30).
   07 EMP-A-CITY    PIC X(20).
   07 EMP-A-ST      PIC X(2).
   07 EMP-A-ZIP     PIC X(9).

 05 EMP-NAME.
   07 EMP-N-LAST    PIC X(15).
   07 EMP-N-FIRST   PIC X(15).
   07 EMP-N-MI      PIC X(1).
```

```
import bea.jam.egenClientBean;
public class EmpRecBean
 extends egenClientBean
{
public EmpRec

   {
    ...
   }
{
```

```
import EmpRecBean;
public class ExtEmpRecBean
  extends EmprecBean
{
  ...
}
```

Figure 1-2 illustrates how your generated application works with JAM.

Figure 1-2 Generated Application Working with JAM



Choosing an eGen Java Application Model

There are four different types or models of Java applications that can be generated by the eGen utility. These are:

- Servlet Only. The servlet-only application is a servlet that presents a simple form and invokes mainframe services directly. This is the simplest model, but it may be unsuitable for production applications.
- Client EJB. The client EJB is a Stateless Session EJB that invokes mainframe services. It may be called by a servlet or other client programs. This is the normal model for building a production application with access to mainframe services. A servlet that invokes the EJB’s methods may be added for testing or demonstration purposes.
- Client Class. The client class is a stand-alone Java class that invokes mainframe services. This class may be built into your own EJB or utilized in some other way within your code.
- Server EJB. The server EJB is a Stateless Session EJB that provides a service to the mainframe.

Choose one of these four model types to use as the basis for your Java application.

Gathering Mainframe Applications Information

Once you have determined which eGen application model you will be using, it is time to gather information about the mainframe application with which JAM will be interacting. The mainframe information gathered will be used to complete the JAM configuration that maps mainframe components to JAM components. COBOL copybooks gathered from the mainframe application will be used to generate Java application code using the eGen utility.

Obtaining Mainframe Services Information

You will need the following mainframe information to develop a Java application that requests a mainframe service:

- Resource names of the services to be requested. The resource name is the name of the mainframe application providing the service.
  - For CICS, the resource name is the equivalent of the CICS program name.
  - For IMS, the resource name is the IMS transaction name.
- COBOL copybooks that define the format of the data sent and/or received from the mainframe. The eGen utility will use these copybooks to generate DataViews.
- A service name that JAM will map to the resource name.

Listing 1-1 shows an example of how a mainframe service is defined. These service definitions can be found in the `JC_REMOTE_SERVICES` section of your JAM gateway configuration file (`jcrmgw.cfg`).

Listing 1-1 Setting a Remote Service in jcrmgw.cfg

```
<ServiceName> RDOM="<Remote Domain>"

              RNAME="<Mainframe Application called>"

              TRANTIME=<transfer time>

              SCHEMA=<Schema Name>
```

You will need the following mainframe information to develop a Java application that responds to requests from the mainframe:

- Resource name of the service to be requested. This name is the service name that the mainframe application (CICS or IMS) invokes to run the Java service.
- COBOL copybooks that define the data layout of data sent and/or received by the mainframe application. The eGen utility will use these copybooks to generate DataViews used by the Java server EJB.

Listing 1-2 shows an example of how a JAM service invoked from the mainframe is defined. These service definitions can be found in the `JC_LOCAL_SERVICES` section of the `jcrmgw.cfg` file.

Listing 1-2 Setting a Local Service (Used by Server EJB) in jcrmgw.cfg

```
<Stateless Session Bean Home> RNAME=”<Resource Name>”
```

Any mainframe service to be requested from JAM must be listed in the JAM gateway configuration file (`jcrmgw.cfg`). These services must be running on your mainframe, either under CICS or IMS. You will need to know the names under which these services are available, and what data formats they require. These data formats will usually be available as COBOL copybooks. If COBOL copybooks are not available, you will need to create them from whatever documentation of the data format you have available.

Mainframe data records are represented in JAM by Java DataView classes. These classes are generated by the eGen utility and provide all of the data translation necessary to communicate with mainframe applications as well as a Java-style access to your data.

The sample files listed in Table 1-1 provide examples of code that can be used to generate Java applications that can request mainframe services. These code example files can be found by extracting them from the `samples.jar` file in your `<JAM Installation>\examples` directory, then looking in the `sample` directory that is created inside the `examples` directory.

Table 1-1 Relevant Example Code from samples.jar File

COBOL Applications	Gateway Config File	COBOL Copybook

`dpldemoc.cbl`
`dpldemor.cbl`
`dpldemou.cbl`
`dpldemod.cbl`
	`jcrmgw1.cfg`	`emprec.cpy`

eGen scripts corresponding to each of the four generation models can also be found in the `samples.jar` file. The applications generated from these scripts are used as examples throughout this guide and *BEA Java Adapter for Mainframe Scenarios*.

Obtaining a COBOL Copybook

A COBOL/CICS or IMS mainframe application typically uses a copybook source file to define its data layout. 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 eGen utility is able to translate most COBOL copybook data types and data clauses into their Java equivalents; however, it is unable to translate some obsolete constructs and floating point data types. For information on COBOL data types that can be translated by the eGen utility, see the “COBOL Datatypes” section of the *BEA Java Adapter for Mainframe Reference Guide*. An eGen utility trial run will reveal any unsupported constructs or data types. If the eGen utility is unable to fully support constructs or data types, it:

- Treats them as alphanumeric data types (if reasonable)
- Ignores them (if their support is unimportant to JAM’s operation)
- Reports them as errors

If the eGen utility reports constructs or data types as errors, you must modify them, so the eGen utility can translate them.

Each copybook’s contents (which define a COMMAREA record) are parsed by the eGen utility, 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.
- 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, then one or more new copybooks may be required. You should keep in mind the COBOL features and data types supported by JAM as you create these copybooks.

Using an Existing COBOL Copybook

When a mainframe application has an existing DPL interface, the data for that interface is probably described in a COBOL copybook. Before using an existing COBOL Copybook, verify that the interface does not use any COBOL features or data types that JAM does not support. To accomplish this task, attempt to process it.

An example COBOL copybook source file is shown in Listing 1-3.

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 1-3 Sample `emprec.cpy` **COBOL Copybook**

```
02     emp-record. (Comment 1)

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

       04     emp-name.
              06     emp-name-last   pic x(15). (Comment 2)
              06     emp-name-first  pic x(15).
              06     emp-name-mi     pic x.

       04     emp-addr. (Comment 3)
              06     emp-addr-street pic x(30).
              06     emp-addr-st     pic x(2).
              06     emp-addr-zip    pic x(9).

* End
```

Table 1-2 refers to the numbered comments in Listing 1-3.

Table 1-2 Script Comments for emprec.cpy

Comment 1	Declaration of a record (group) data item.
Comment 2	An elementary item. This is the base level of the data structure.
Comment 3	An aggregate item. This is the intermediate level of the data structure.

Writing an eGen COBOL Script

After you have gathered information about the mainframe applications and have decided on an eGen Java application model for it, you are ready to write an eGen COBOL script. This eGen script and the COBOL copybook that describes your data structure will be processed by the eGen utility to generate the basis for your custom Java application.

An eGen COBOL script has two sections. These are:

- DataView. The DataView section of the script generates Java DataView code from a COBOL copybook. The class file compiled from the generated code extends the Java DataView class.
- Java application. The Java application section of the script generates the Java application code.

The eGen scripts that can be used to generate example applications corresponding to each of the four generation models can be found in the `samples.jar` file. The `samples.jar` file can be found in the `<JAM Installation>\examples` directory. The application code generated from these scripts is used as examples throughout this guide and *BEA Java Adapter for Mainframe Scenarios*.

Writing the DataView Section of an eGen COBOL Script

The eGen utility parses a COBOL copybook and generates Java DataView code 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 example shown in Listing 1-4. This section is only the first section of the eGen script. Application code is generated by the second section.

Listing 1-4 Sample DataView Section of an eGen COBOL Script

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

Analyzing the parts of this line of code, we see that **generate view** tells the eGen utility to generate a Java DataView code file. `sample.EmployeeRecord` tells the eGen utility to call the DataView file EmployeeRecord and place it in a package called sample. The code `from emprec.cpy` tells the eGen utility to form the EmployeeRecord DataView file from the COBOL copybook emprec.cpy.

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.

Listing 1-5 Sample DataView Section with Codepage Specified

```
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:

Listing 1-6 Sample DataView Section Supporting XML

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

Additional files generated for XML support are listed in Table 1-3.

Table 1-3 Additional Files for DataView XML Support.

File Name	File Purpose
classname.dtd	XML DTD for XML messages accepted and produced by this DataView.
classname.xsd	XML schema for XML messages accepted and produced by this DataView.

Writing the Application Section of an eGen COBOL Script

Now that you have written the DataView section of the script and have determined which application model that you want to generate, refer to the section from the following list for instructions on writing the script and implementing the model you have chosen:

- "Generating a Servlet-Only Application"
- "Generating a Client Enterprise Java Bean-based Application"
- "Generating a Server Enterprise Java Bean-based Application"
- "Generating a Stand-alone Client Application"

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

Processing eGen Scripts with the eGen Utility

After you have written your eGen COBOL script, you must process it. Processing the eGen COBOL script generates Java DataView and application code. This Java code must be compiled and deployed. Although processing the eGen COBOL script into DataView and application code is usually performed in one step, it will be explained in two steps, so the actual code generated can be analyzed in greater detail.

Creating an Environment for Generating and Compiling the Java Code

Before you create an environment for generating and compiling your Java application code, you must already have set up your environment as explained in the BEA WebLogic Server documentation and the *BEA Java Adapter for Mainframe Installation Guide*.

When you process the eGen COBOL scripts and compile the generated Java code, you must have access to the Java classes and applications used in the code generation and compilation processes. Adding the correct elements to your `CLASSPATH` and `PATH` environment variables provides access to the necessary Java classes and applications.

For the eGen utility:

- Add `<JAM Installation Directory>\lib\jam.jar` to your `CLASSPATH`.
- Add `<JAM Installation Directory>\bin` to your `PATH`.

For compilation:

- Add `<JAM Installation Directory>\lib\jam.jar` to your `CLASSPATH`.
- Add `<WLS_HOME>\lib\weblogic.jar` to your `CLASSPATH`.
- Add path of your `DataView` class files to your `CLASSPATH` . You will need access to these classes when you compile your Java application code.

Note: UNIX users must use “/” instead of “\” when adding directory paths as specified above.

Generating the Java DataView Code

The script in Listing 1-7 specifies that the COBOL copybook file named `emprec.cpy` is parsed, and that the Java DataView source file named `EmployeeRecord.java` is generated from it. Also, this file is added to package `sample`. In other words, when this script is processed by the eGen utility, a file named `EmployeeRecord.java` is generated, and it contains the definition of class `EmployeeRecord` in package `sample`.

(If you are referring to the sample files that can be extracted from `samples.jar`, note that this file is contained in a directory called `sample`.) The `EmployeeRecord` class is an instance of the DataView class.

Listing 1-7 Sample DataView Section of `emprec.egen` Script

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

If you saved this script in a file named `emprec.egen`, the following shell command parses the copybook file named `emprec.cpy` and generates the `EmployeeRecord.java` source file in the current directory:

Listing 1-8 Sample Copybook Parse Command

```
egencobol emprec.egen
```

If no error or warning messages are issued, the copybook is compatible with JAM and the source files are created. If you are generating DataView code using the sample code provided in the `samples.jar` file, you will notice that no application source files are generated by processing the `emprec.egen` script. No application source files are generated because there are no application generating commands in this script.

Note: Refer to the "Error Messages" section of the *BEA Java Adapter for Mainframe Reference Guide* for suggestions on resolving any problems encountered.

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

Listing 1-9 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
```

Table 1-4 refers to the numbered comments in Listing 1-9.

Table 1-4 Script Comments for EmployeeRecord.java

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.

Generating the Java Application Code

The Java application code can be generated at the same time that you generate the Java DataView code. To generate Java application code, the eGen COBOL script that you process must contain instructions for generating the Java application along with the instructions for generating the DataView code.

Referring to the sample files in `samples.jar`, the following command generates `EmployeeRecord.java` and `SampleServlet.java`. `EmployeeRecord.java` is the DataView file, and `SampleServlet.java` is the application file.

```
> egencobol empservlet.egen
```

Special Considerations for Compiling the Java Code

You must compile the Java code generated by the eGen utility. However, there are some special circumstances to consider. Because the application code is dependent on the DataView code, you must compile the DataView code and make sure that the resulting DataView class files are in your environment's `CLASSPATH` before compiling your application code. You must make sure that all of the DataView class files can be referenced by the application code compilation.

For example, the compilation of `EmployeeRecord.java` results in four class files:

- `EmployeeRecord.class`
- `EmployeeRecord$EmpRecord1V.class`
- `EmployeeRecord$EmpRecord1V$EmpName3V.class`
- `EmployeeRecord$EmpRecord1V$EmpAddr7V.class`

All of these class files are used when compiling your application code.

Deploying Applications

Deployment is the process of implementing servlets and/or EJBs on WebLogic Server. Application deployment in WebLogic Server has evolved to the J2EE standard for web application deployment.

The following information is not intended to specifically describe how applications are deployed in WebLogic Server. For specific information, refer to Quick Start information and detailed documentation for deploying applications in the WebLogic Server 6.0 online documentation at:

```
http://edocs/wls/docs60/quickstart/quick_start.html
http://edocs/wls/docs60/servlet/admin.html#156888
http://edocs/wls/docs60/ejb/EJB_deployover.html
```

Deploying a JAM eGen Servlet (Quick-Start Deployment)

The basic JAM eGen servlet is deployed like any other WebLogic servlet. The configuration for the eGen servlet is stored in the `web.xml` file in an applications directory associated with a domain. The basic default configuration can be found in the following directory:

```
<bea_home>/<wls_home>
      /config/mydomain/applications/DefaultWebApp_myserver/WEB-INF/web.xml
```

For the `SampleServlet` (generated by the `egencobol empservlet.egen` command), add the `classes` and `sample` directories, so the directory structure looks like the following:

```
<bea_home>/<wls_home>/config/mydomain
      /applications/DefaultWebApp_myserver/WEB-INF/classes/sample
```

The eGen `SampleServlet` and `EmployeeRecord` class, which are the result of compiling the `*.java` files generated by the eGen utility, should be placed in the `sample` directory:

```
<bea_home>/<wls_home>/config/mydomain
      /applications/DefaultWebApp_myserver/WEB-INF/classes/sample
```

SampleServlet can be configured with an XML entry (added to web.xml) similar to the one shown in Listing 1-10:

Listing 1-10 XML Entry to Configure the SampleServlet Servlet

```
<web-app>
       <servlet>
              <servlet-name>
                     SampleServlet
              </servlet-name>
              <servlet-class>
                     sample.SampleServlet
              </servlet-class>
       </servlet>
       <servlet-mapping>
              <servlet-name>
                     SampleServlet
              </servlet-name>
              <url-pattern>
                     /SampleServlet/*
              </url-pattern>
       </servlet-mapping>
</web-app>
```

SampleServlet can then by invoked by entering the following URL in the location field of your web browser:

```
http://<host>:<port>/SampleServlet
```

If WebLogic Server is running on your local machine and you used the default port (7001) when you installed WebLogic Server, SampleServlet can be invoked by the following URL:

```
http://localhost:7001/SampleServlet
```

Deploying a JAM eGen EJB

A JAM eGen EJB (client or server) is deployed like any other WebLogic EJB. The following instructions and examples are provided as an aid:

1. Build your EJB deployment JAR file. Listing 1-11 will build the client EJB deployment JAR file from the components generated by the `empclient.egen` eGen COBOL script and `emprec.cpy`.

Listing 1-11 Script for Building empclientbean.jar

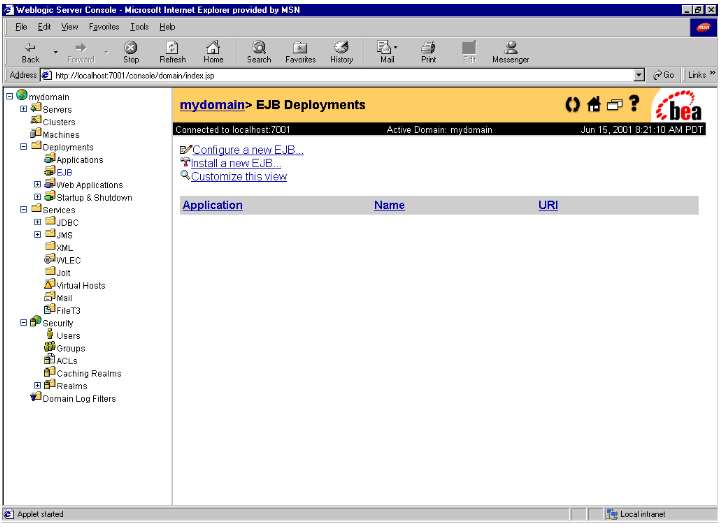
```
@rem  --- Adjust these variables to match your environment
-----------------
set TARGETJAR=empclientbean.jar
set JAVA_HOME=c:\bea\jdk130
set WL_HOME=c:\bea\wlserver6.0sp1
set JAM_HOME=c:\bea\wljam4.2
@rem  ------ end of Adjustable variables
------------------------------------

set JAMJARS=%JAM_HOME%\lib\jam.jar
set CLASSPATH=%JAM_HOME%\lib\jam.jar;%JAM_HOME%\lib\tools.jar;
%WL_HOME%\lib\weblogic.jar
set PATH=%JAVA_HOME%\bin;%JAVA_HOME%\lib;%PATH%

@rem  Create the build directory, and copy the deployment
@rem  descriptors into it.
@rem  You should have already run your egen script so your xml files
@rem  are already built.

md build build\META-INF
copy SampleClient-jar.xml ejb-jar.xml
copy wl-SampleClient-jar.xml weblogic-ejb-jar.xml
copy *.xml build\META-INF

@rem  Compile ejb classes into the build directory (jar preparation)
javac -d build -classpath %CLASSPATH%  *.java

@rem  Make a standard ejb jar file, including XML deployment
@rem  descriptors
cd build
jar cvf std_%TARGETJAR% META-INF sample
cd ..

@rem  Run ejbc to create the deployable jar file

java -classpath %CLASSPATH% -Dweblogic.home=%WL_HOME%
weblogic.ejbc -compiler javac build\std_%TARGETJAR% %TARGETJAR%
```

2. Deploy the EJB in BEA WebLogic Server by configuring it as a new EJB in the BEA WebLogic Server Admin Console. Configure this new EJB as follows:

   a. Click the **EJB** icon under **Deployments**.

      The **EJB Deployments** screen appears (see Figure 1-3).

Figure 1-3 Configuring a New EJB, empclientbean.jar



http://localhost:7001/console/domain/index.jsp

mydomain> EJB Deployments

Connected to localhost:7001 Active Domain: mydomain Jun 15, 2001 8:21:10 AM PDT

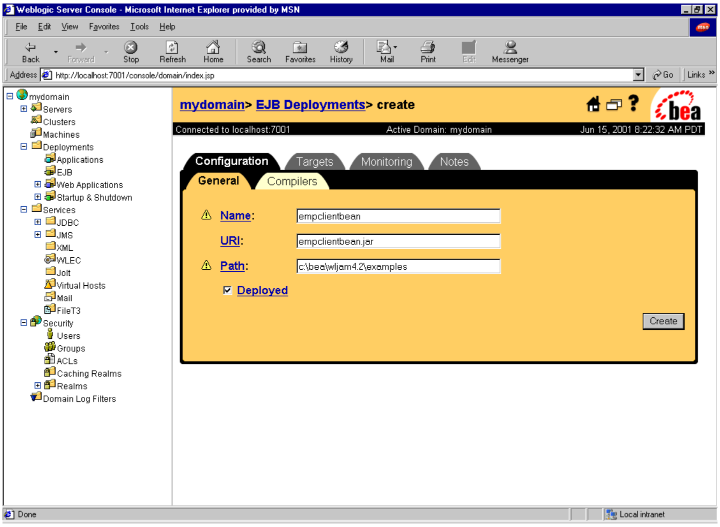
Configure a new EJB...
Install a new EJB...
Customize this view

Application	Name	URI

   b. Click the **Configure a new EJB** link.

      The **EJB Deployments Create** screen appears (see Figure 1-4).

Figure 1-4 New EJB Configuration Screen



mydomain> EJB Deployments> create

Connected to localhost:7001 Active Domain: mydomain Jun 15, 2001 8:22:32 AM PDT

Name: empclientbean

URI: empclientbean.jar

Path: c:\bea\wljam4.2\examples

☑ Deployed

c. Enter the name of your EJB in the **Name:** field, the EJB Deployment JAR file in the **URI:** field, and the path to the EJB Deployment JAR file in the **Path:** field. Make sure that the **Deployed** checkbox is checked. Then, click **Create**.

Your JAM eGen EJB is now deployed.

Providing OS/390 Mainframe Access with No Data Translation

JAM may be used for OS/390 mainframe access without performing data translation. All client-side code generated by the eGen utility uses the EgenClient class to access the gateway. This class provides a raw-byte interface to the gateway. Use of the EgenClient class for gateway access has the following advantages and disadvantages:

- Advantages:

a. No translation infrastructure overheads are incurred

   b. Simple interface

   c. JAM's client diagnostic features may be used

- Disadvantages:

   a. Not used for requests inbound from the mainframe

Use the interface of the EgenClient class (shown in Listing 1-12) to perform raw mainframe requests.

Listing 1-12 Interface for Performing Raw Mainframe Requests

```
package com.bea.jam.egen;
import com.bea.sna.jcrmgw.snaException;
public class EgenClient
{
  public EgenClient();
  public byte[] callService(String service, byte[] in)
  throws snaException, java.io.IOException;
}
```

Note: If an example of the proper use of this class is desired, the eGen utility may be used to generate a client class.

Using Client Diagnostic Features with WebLogic Server 6.0

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 settings listed in Table 1-5 to the java statement at the end of your `startWebLogic.cmd` file for the BEA WebLogic Server domain that you are currently running.

Table 1-5 Client Diagnostic Settings

bea.jam.client.trace.enable	Set to "true" to enable tracing of client requests.
bea.jam.client.trace.codepage	Set to the name of a codepage to be used for the character portion of the trace dump.
bea.jam.client.loopback	Set to "true" to bypass the gateway & simply loop the request bytes back to the client.
bea.jam.client.stub	Set to the full name of a class to be used as a gateway stub.

Listing 1-13 provides an example in **bold** of the changes that need to be made to the java statement in the `startWebLogic.cmd` file necessary to enable the client diagnostic loopback feature. This file can be found in the `%WLS_HOME%\config\<domain>` directory. The java statement can be found near the end of the file.

Listing 1-13 startWebLogic.cmd Loopback Example

```
                         ...
"%JAVA_HOME%\bin\java" -hotspot -ms64m -mx64m -classpath
%CLASSPATH% -Dweblogic.Domain=mydomain
-Dbea.jam.client.loopback=true -Dweblogic.Name=myserver
"-Dbea.home=g:\bea"
"-Djava.security.policy==g:\bea\wlserver6.0sp1/lib/weblogic.polic
y" -Dweblogic.management.password=%WLS_PW% weblogic.Server
                         ...
```

Client Traffic Tracing

When client traffic tracing is enabled, all requests from eGen clients are dumped to the WebLogic console in hexadecimal and EBCDIC characters. Listing 1-14 shows an example of an eGen client dump.

Listing 1-14 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 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 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 00 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 are usually 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 the client loopback feature is enabled, all requests receive a response that is exactly equal to the request data. Note that this loopback response is accomplished while the data is in mainframe format. If a service accepts one DataView subclass and returns a different one, a conversion failure in trying to construct the resulting DataView subclass may occur.

When the client loopback feature is enabled, no gateway is required and the `gwboot` startup class does not need to be configured.

Client Stub Operation

The client stub operation 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 `CLASSPATH`.
- 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.

What Do I Do Next?

Refer to the chapter from the following list that corresponds to the Java application model you have chosen for your Java application:

- “Generating a Servlet-Only Application”
- “Generating a Client Enterprise Java Bean-based Application”
- “Generating a Server Enterprise Java Bean-based Application”
- “Generating a Stand-alone Client Application”

Also, refer to *BEA Java Adapter for Mainframe Scenarios* for detailed examples of some of the application models discussed in this guide.

CHAPTER

2 Generating a Servlet-Only Application

A JAM servlet-only application is a Java servlet that executes within BEA WebLogic Server. The application 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 JAM Gateway)

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

Action List

Before you generate a servlet-only application, see the following action list and refer to the appropriate information sources.

	Your action...	**Refer to...**
1	Complete all prerequisite tasks.	“Prerequisites”
2	Review the general steps for building a Java application.	“Generating a Java Application with the eGen COBOL Code Generator”
3	Review an example of a script for generating a servlet-only application.	“Writing an eGen COBOL Servlet Script”
4	Review script processing and sample script commands.	“Processing a Script to Generate Your Application Source Files”
5	Review the generated files.	“Reviewing the Generated Files”
6	Customize the application.	“Customizing a Servlet-Only JAM Application”
7	Proceed to the next set of instructions.	“What Do I Do Next?”

Prerequisites

Before you start programming your servlet-only application, you should complete the following tasks:

	Your action...	**Refer to...**
1	Install your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Configure your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*

Writing an eGen COBOL Servlet Script

In order to produce a servlet-only application, create an `*.egen` script file and use the eGen COBOL Code Generator (also called the eGen utility) to generate your typed data record (DataView), and Servlet code. Your DataView files must describe the mainframe services accessed, the browser pages produced, and the servlets that produce them. Service definitions look like the following listing.

Listing 2-1 Sample Service Definition in eGen COBOL Script

```
service sampleCreate
       accepts EmployeeRecord
       returns EmployeeRecord
```

Listing 2-1 defines a service named `sampleCreate` that accepts an input buffer of type `EmployeeRecord` and returns an output buffer of type `EmployeeRecord.` This service is configured by an entry in the `SERVICES` section of the `jcrmgw.cfg` file. Listing 2-2 provides an example of an entry that configures the service `sampleCreate`. This example configures the `sampleCreate` function to run on the CICS410 remote domain (`RDOM=”CICS410”`). The `sampleCreate` service invokes the CICS program named `DPLDEMOC` (`RNAME=”DPLDEMOC”`), and it uses the `sample.EmployeeRecord` schema. The files that actually define the schema are `EmployeeRecord.java`, `EmployeeRecord.xsd` and `EmployeeRecord.dtd`. These files are generated by the eGen utility when `support xml` is appended on the generate view code line of the script.

Listing 2-2 Sample jcrmgw.cfg Entry for Service sampleCreate

```
sampleCreate  RDOM=”CICS410”
              RNAME=”DPLDEMOC”
              TRANTIME=10000
              SCHEMA=sample.EmployeeRecord
```

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

Listing 2-3 Sample Page Definition

```
page initial “Initial Page”
{
       view EmployeeRecord

       buttons
       {
              “Create”
                     service (“sampleCreate”)
                     shows fullPage
       }
}
```

This listing defines an HTML page named `initial`, with a text title of “Initial Page”, that displays an `EmployeeRecord` record object as an HTML form. It also specifies that the form has a button labeled “Create”. When the button is pressed, the service `sampleCreate` 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 `fullPage` page is used to display the results.

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

Listing 2-4 Sample servlet Definition

```
servlet sample.SampleServlet shows initial
```

This listing defines an application servlet class named `SampleServlet`, and specifies that it displays the HTML page named “Initial Page” as its initial display page.

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

Listing 2-5 Sample Servlet-Only Script

```
#---------------------------------------------------------
# empservlet.egen
#   JAM script for a servlet-only application.
#
```

```
#  $Id: empservlet.egen,v 1.2 2000/01/25  18:34:14 david Exp$
#---------------------------------------------------------------

# DataViews (typed data records)

view sample.EmployeeRecord (Comment 1)
   from emprec.cpy support xml

# Services

service sampleCreate (Comment 2)
   accepts EmployeeRecord
   returns EmployeeRecord

service sampleRead (Comment 2)
   accepts EmployeeRecord
   returns EmployeeRecord

service sampleUpdate (Comment 2)
   accepts EmployeeRecord
   returns EmployeeRecord

service sampleDelete (Comment 2)
   accepts EmployeeRecord
   returns EmployeeRecord

# Servlet HTML pages

page initial “Initial page” (Comment 3)
{
   view EmployeeRecord (Comment 4)

   buttons
   {
          “Create” (Comment 5)
                 service (“sampleCreate”)
                 shows fullPage

          “Read” (Comment 5)
                 service (“sampleRead”)
                 shows fullPage
   }
}

page fullPage “Complete page”
{
   view EmployeeRecord

   buttons
   {
          “Create”
                 service (“sampleCreate”)
                 shows fullPage

          “Read”
                 service (“sampleRead”)
                 shows fullpage

          “Update”
                 service (“sampleUpdate”)
                 shows fullpage

          “Delete”
                 service (“sampleDelete”)
                 shows fullpage
   }
}

# Servlets

servlet sample.SampleServlet (Comment 6)
   shows initial

# End
```

Table 2-1 refers to the numbered comments in Listing 2-5.

Table 2-1 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 Generate Your Application Source Files

To process the script, issue the command in Listing 2-6. The `egencobol` command involves the JVM and is equivalent to `java com.bea.jam.egen.EgenCobol empservlet.egen`.

Listing 2-6 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...
```

Reviewing the Generated Files

The `empservlet.egen` script command generates the following files.

Table 2-2 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

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

Listing 2-7 Sample `SampleServlet.java` **Contents**

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

package sample;

//  Imports

import javax.servlet.http.HttpServlet;
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
```

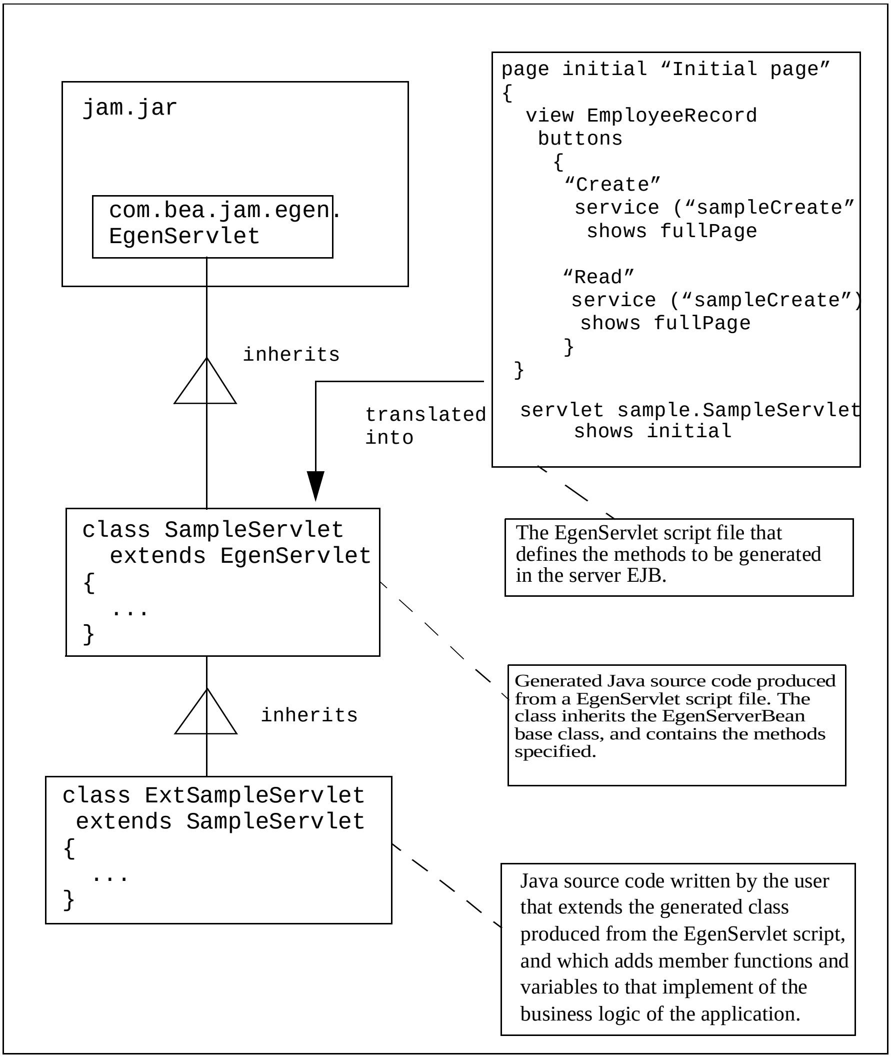
Customizing a Servlet-Only JAM Application

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 2-1 The JAM Servlet Class Hierarchy



```
page initial "Initial page"
{
  view EmployeeRecord
   buttons
     {
      "Create"
       service ("sampleCreate"
        shows fullPage

      "Read"
       service ("sampleCreate")
        shows fullPage
      }
 }

 servlet sample.SampleServlet
      shows initial
```

```
class SampleServlet
  extends EgenServlet
{
   ...
}
```

```
class ExtSampleServlet
 extends SampleServlet
{
   ...
}
```

The EgenServlet script file that defines the methods to be generated in the server EJB.

Generated Java source code produced from a EgenServlet script file. The class inherits the EgenServerBean base class, and contains the methods specified.

Java source code written by the user that extends the generated class produced from the EgenServlet script, and which adds member functions and variables to that implement of the business logic of the application.

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 2-8 `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=reqgetSession(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.

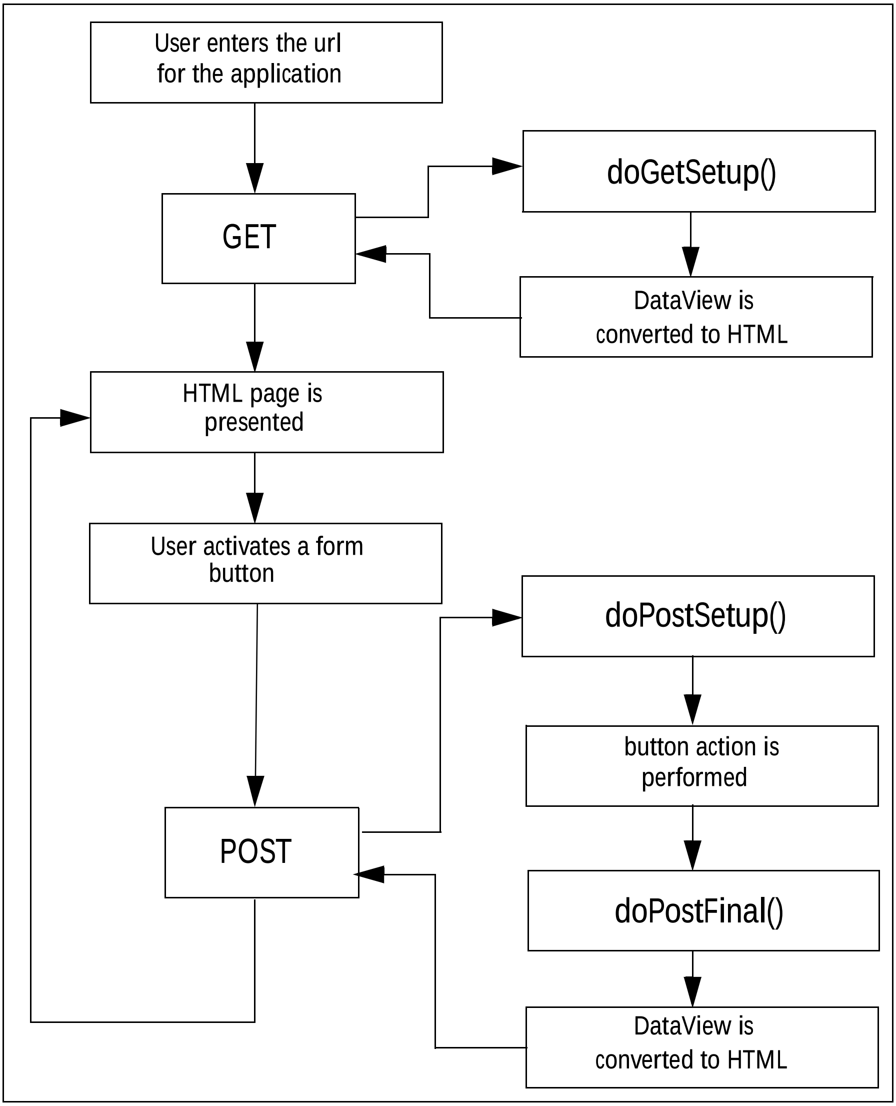
  This function occurs prior to the presentation of the HTML page to the user's browser after activating a form button. Any changes made to the DataView object will be reflected in the contents of the HTML page.

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 2-2 User Browser-Session Flowchart


User enters the url for the application
GET
doGetSetup()
DataView is converted to HTML
HTML page is presented
User activates a form button
doPostSetup()
button action is performed
POST
doPostFinal()
DataView is converted to HTML

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 a trace of the execution stack that was in effect at the time that the exception was thrown.

Listing 2-9 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 it has been written, the `ExtSampleServlet` class and the other servlet Java source files must be compiled and deployed in the same manner as other servlets.

What Do I Do Next?

Refer to *BEA Java Adapter for Mainframe Scenarios* for detailed examples of some of the application models discussed in this guide.

CHAPTER

3 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.

Action List

Before you build a Client Enterprise Java Bean-based application, see the following action list and refer to the appropriate information sources.

	Your action...	Refer to...
1	Complete all prerequisite tasks.	“Prerequisites”

	Your action...	Refer to...
2	Review the general steps for building a Java application	“Generating a Java Application with the eGen COBOL Code Generator”
3	Review an example of a script for generating a client EJB application	“Components of an eGen COBOL Client EJB Script”
4	Review script processing and sample script commands	“Processing the Script”
5	Review the generated files	“Working with Generated Files”
6	Customize the application	“Customizing an Enterprise Java Bean-Based Application”
7	Proceed to the next set of instructions.	“What Do I Do Next?”

Prerequisites

Before you start programming your Client Enterprise Java Bean-based application, you should complete the following tasks:

	Your action...	Refer to...
1	Install your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Configure your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*

Components of an eGen COBOL Client EJB Script

In order to produce an EJB-based application, the script file that defines your DataViews must be edited to describe both the mainframe services accessed and the EJB that will access them. A service description might look like the listing in Listing 3-1.

Listing 3-1 Sample `service` **Description**

```
service sampleCreate

       accepts EmployeeRecord
       returns EmployeeRecord
```

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

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

Listing 3-2 Sample `getSalary` **Service Definition**

```
client ejb MyEJBName MyEJBHome
{
       method newEmployee is service sampleCreate
}
```

This listing defines a Java bean class named `MyEJBName` with a method named `newEmployee`. The method corresponds to service name `sampleCreate`. 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 3-3 Sample Client EJB Script

```
#----------------------------------------------------------
# empclient.egen
#   JAM script for an employee record.
#
#  $Id: empclient.egen,v 1.1 2000/01/25  18:34:14 david Exp$
#---------------------------------------------------------------

# DataViews (typed data records)

view sample.EmployeeRecord (Comment 1)
   from emprec.cpy

# Services

service sampleCreate (Comment 2)
   accepts EmployeeRecord
   returns EmployeeRecord

service sampleRead (Comment 2)
   accepts EmployeeRecord
   returns EmployeeRecord

# Clients and servers

client ejb sample.SampleClient my.sampleBean (Comment 3)
{
   method newEmployee (Comment 4)
          is service sampleCreate

   method readEmployee (Comment 4)
          is service sampleRead
}

# End
```

Table 3-1 refers to the numbered comments in Listing 3-3.

Table 3-1 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

Issue the following command to process the script.

Listing 3-4 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

The `empclient.egen` script command generates the following files.

Table 3-2 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 3-5 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
```

Table 3-3 refers to the numbered comments in Listing 3-5.

Table 3-3 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

Listing 3-6 shows the contents of the generated `SampleClientBean.java` source file.

Listing 3-6 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
```

Table 3-4 refers to the numbered comments in Listing 3-6.

Table 3-4 Script Comments

Comment 1	Defines an EJB client bean.
Comment 2	The methods convert a raw COMMAREA into a Java DataView object.

SampleClientHome.java Source File

Listing 3-7 shows the contents of the generated `SampleClientHome.java` deployment descriptor file.

Listing 3-7 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
```

Table 3-5 refers to the numbered comments in Listing 3-7.

Table 3-5 Script Comments

Comment 1	Defines an EJB home interface.

SampleClient-jar.xml Source File

Listing 3-8 shows the contents of the generated `SampleClient-jar.xml` deployment descriptor file.

Listing 3-8 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

Listing 3-9 shows the contents of the `wl-SampleClient-jar.xml` source file. To use this file, copy it to `weblogic-ejb-jar.xml`.

Listing 3-9 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>
```

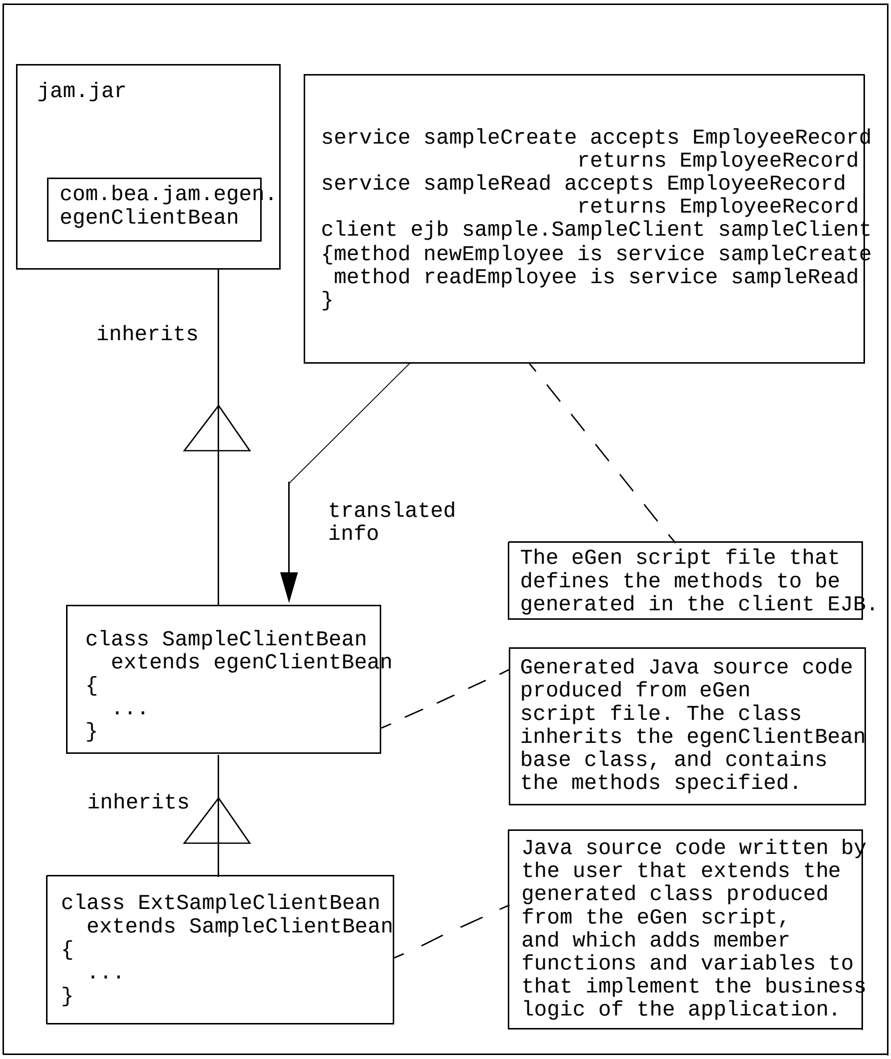
Customizing an Enterprise Java Bean-Based Application

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 3-1 The JAM Client EJB Class Hierarchy



```
service sampleCreate accepts EmployeeRecord
                     returns EmployeeRecord
service sampleRead accepts EmployeeRecord
                     returns EmployeeRecord
client ejb sample.SampleClient sampleClient
{method newEmployee is service sampleCreate
 method readEmployee is service sampleRead
}
```

The eGen script file that defines the methods to be generated in the client EJB.

Generated Java source code produced from eGen script file. The class inherits the egenClientBean base class, and contains the methods specified.

Java source code written by the user that extends the generated class produced from the eGen script, and which adds member functions and variables to that implement the business logic of the application.

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 Listing 3-10, 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 3-10 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(),
       // ejbPassiveate(), 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 Listing 3-11, adds the methods specific to this EJB.

Listing 3-11 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
```

Listing 3-12 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 3-12 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  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.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 it has been written, the `ExtSampleClientBean` class 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 WebLogic Server contains a build script for reference.

What Do I Do Next?

Refer to *BEA Java Adapter for Mainframe Scenarios* for detailed examples of some of the application models discussed in this guide.

CHAPTER

4 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.

Action List

Before you build a Server Enterprise Java Bean-based application, see the following action list and refer to the appropriate information sources.

	Your action...	Refer to...
1	Complete all prerequisite tasks.	“Prerequisites”

	Your action...	Refer to...
2	Review the general steps for building a Java application	"Generating a Java Application with the eGen COBOL Code Generator"
3	Review an example of a script for generating a server EJB application	"Components of an eGen COBOL Server EJB Script"
4	Review script processing and sample script commands	"Processing the Script"
5	Review the generated files	"Working with Generated Files"
6	Customize the application	"Customizing a Server Enterprise Java Bean-Based Application"
7	Proceed to the next set of instructions.	"What Do I Do Next?"

Prerequisites

Before you start programming your Server Enterprise Java Bean-based application, you should complete the following tasks:

	Your action...	Refer to...
1	Install your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Configure your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*

Components of an eGen COBOL Server EJB Script

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

Listing 4-1 Sample Server EJB Script

```
#--------------------------------------------------------
# empserver.egen
#   JAM script for an employee record.
#
# $Id: empserver.egen, v 1.1 2000/01/21 23:20:40
#-------------------------------------------------------------

# DataViews (typed data records)

view sample.EmployeeRecord (Comment 1)
   from emprec.cpy

# Clients and servers (Comment 2)

server ejb sample.SampleServer my.sampleServer (Comment 3)
{
   method newEmployee (EmployeeRecord)(Comment 4)
          returns EmployeeRecord
}

# End
```

Table 4-1 refers to the numbered comments in Listing 4-1.

Table 4-1 Script Comments

Comment 1	Defines a DataView class, specifying its corresponding copybook source file and its package name.
Comment 2	Defines a server EJB class.

Table 4-1 Script Comments

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

Issue the following command to process the script.

Listing 4-2 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...
```

Working with Generated Files

The `empserver.egen` script command generates the following files.

Table 4-2 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.

Table 4-2 Sample Script Generated Files

File	Content
`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 4-3 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 4-4 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 4-5 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 deployment descriptor file.

Listing 4-6 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 4-7 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
```

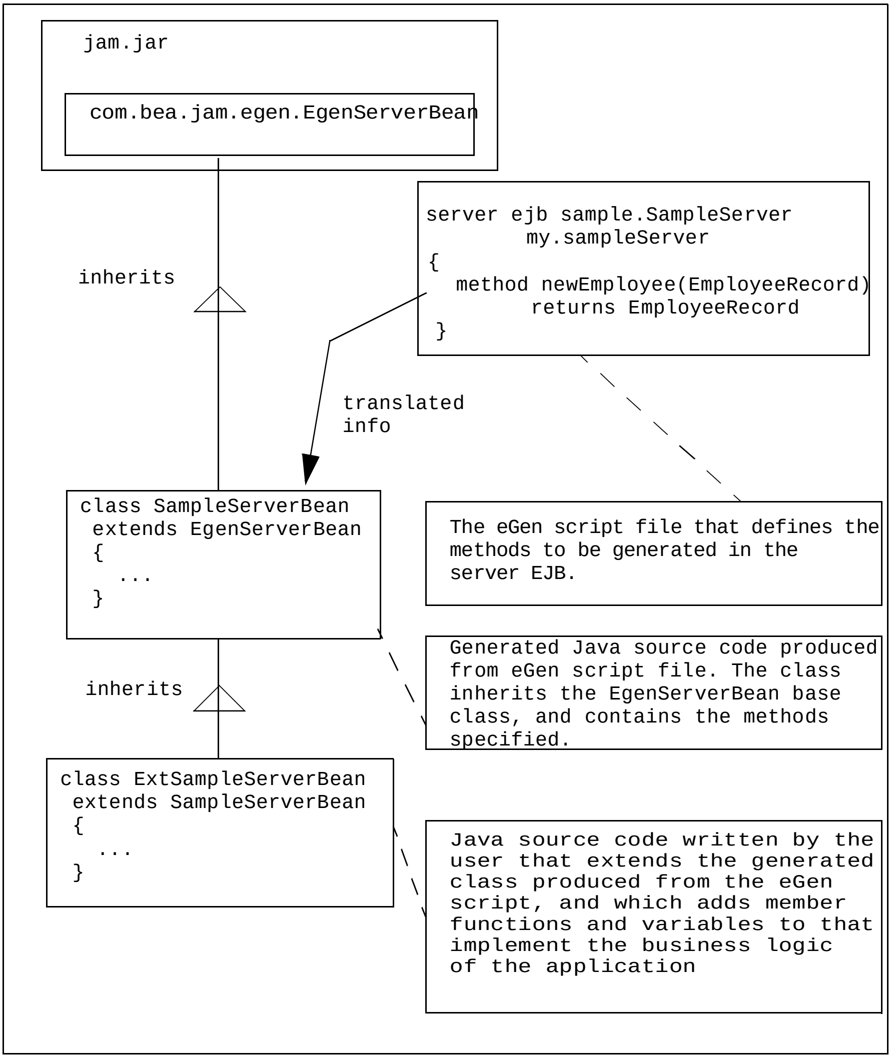
Customizing a Server Enterprise Java Bean-Based Application

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 4-1 The JAM Server EJB Class Hierarchy


jam.jar
com.bea.jam.egen.EgenServerBean
server ejb sample.SampleServer
my.sampleServer
{
method newEmployee(EmployeeRecord)
returns EmployeeRecord
}
inherits
translated info
class SampleServerBean
extends EgenServerBean
{
...
}
The eGen script file that defines the methods to be generated in the server EJB.
Generated Java source code produced from eGen script file. The class inherits the EgenServerBean base class, and contains the methods specified.
inherits
class ExtSampleServerBean
extends SampleServerBean
{
...
}
Java source code written by the user that extends the generated class produced from the eGen script, and which adds member functions and variables to that implement the business logic of the application

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 4-8 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 4-9 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 4-10 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 it has been written, the `ExtSampleServerBean` class 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 WebLogic Server contains a build script for reference.

What Do I Do Next?

Refer to *BEA Java Adapter for Mainframe Scenarios* for detailed examples of some of the application models discussed in this guide.

CHAPTER

5 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.

Action List

Before you build a stand-alone application, see the following action list and refer to the appropriate information sources.

	Your action...	Refer to...
1	Complete all prerequisite tasks.	“Prerequisites”
2	Review the general steps for building a Java application	“Generating a Java Application with the eGen COBOL Code Generator”
3	Review an example of a script for generating a stand-alone Java application	“Components of an eGen COBOL Stand-alone Application Script”

	Your action...	Refer to...
4	Review script processing and sample script commands	"Processing a Script"
5	Review the generated files	"Working with Generated Files"
6	Customize the application	"Customizing a Stand-Alone Java Application"
7	Proceed to the next set of instructions.	"What Do I Do Next?"

Prerequisites

Before you start programming your stand-alone application, you should complete the following tasks:

	Your action...	Refer to...
1	Install your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Configure your computer systems, Windows/UNIX and mainframe, to meet your requirements.	*BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*

Components of an eGen COBOL Stand-alone Application Script

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

Listing 5-1 Sample Stand-Alone Client Class Script

```
#--------------------------------------------------------
# empclass.egen
#   JAM script for an employee record.
#
# $Id: empclass.egen, v 1.1 2000/01/21 23:20:40
#-------------------------------------------------------------

# DataViews (typed data records)

view sample.EmployeeRecord (Comment 1)
    from emprec.cpy

 # Services

 service sampleCreate (Comment 2)
    accepts EmployeeRecord
    returns EmployeeRecord

 service sampleRead (Comment 2)
    accepts EmployeeRecord
    returns EmployeeRecord

 service sampleUpdate (Comment 2)
    accepts EmployeeRecord
    returns EmployeeRecord

 service sampleDelete (Comment 2)
    accepts EmployeeRecord
    returns EmployeeRecord

 # Clients and servers

 client class sample.SampleClass (Comment 3)
```

```
{
   method newEmployee (Comment 4)
          is service sampleCreate

   method readEmployee (Comment 4)
          is service sampleRead
}

# End
```

Table 5-1 refers to the numbered comments in Listing 5-1.

Table 5-1 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

Issue the following command to process the script.

Listing 5-2 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...
```

Working with Generated Files

This script command generates the following files.

Table 5-2 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 5-3 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
```

Table 5-3 refers to the numbered comments in Listing 5-3.

Table 5-3 Script Comments

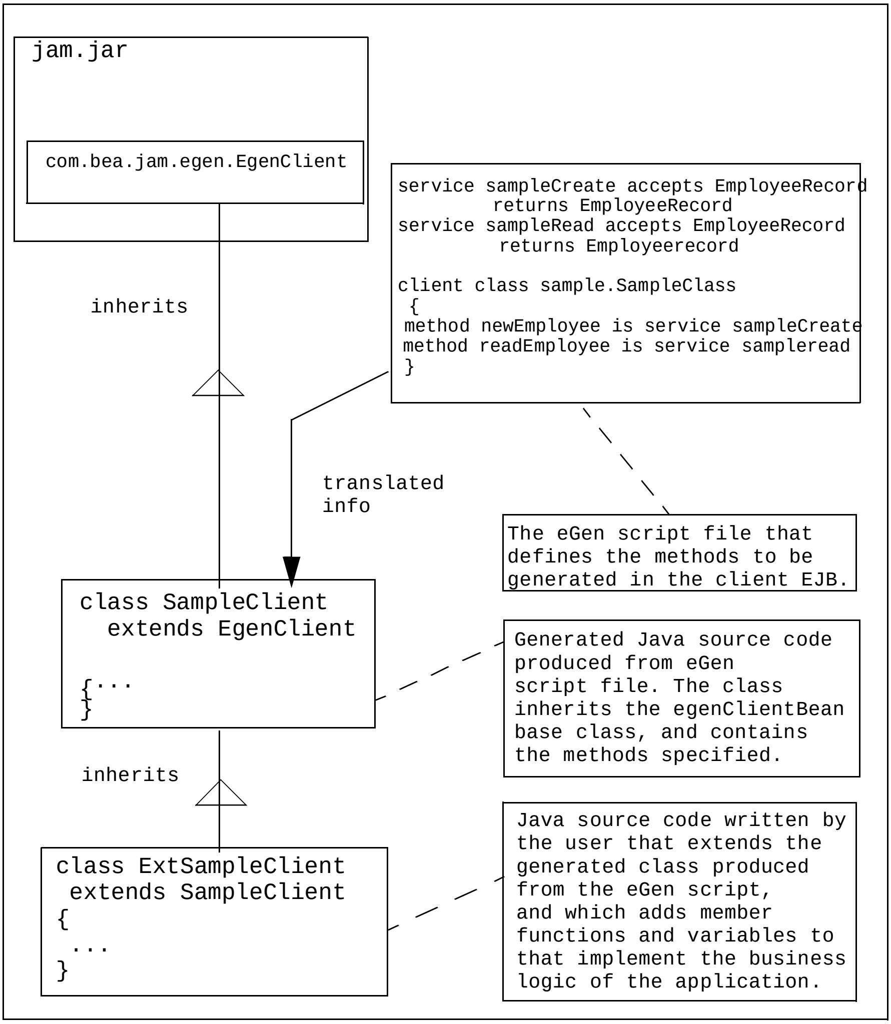
Comment 1	The package name is defined in the eGen script.
Comment 2	The data record is encapsulated in a class that extends the `EgenClient` class.
Comment 3	The methods convert a raw COMMAREA into a Java DataView object.

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 5-1 The JAM Client EJB Class Hierarchy


jam.jar
com.bea.jam.egen.EgenClient
inherits
service sampleCreate accepts EmployeeRecord
returns EmployeeRecord
service sampleRead accepts EmployeeRecord
returns Employeerecord
client class sample.SampleClass
{
method newEmployee is service sampleCreate
method readEmployee is service sampleread
}
translated info
The eGen script file that defines the methods to be generated in the client EJB.
class SampleClient extends EgenClient
{...
}
Generated Java source code produced from eGen script file. The class inherits the egenClientBean base class, and contains the methods specified.
inherits
class ExtSampleClient extends SampleClient
{
...
}
Java source code written by the user that extends the generated class produced from the eGen script, and which adds member functions and variables to that implement the business logic of the application.

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 5-4 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 5-5 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. Your class may also add additional methods, if desired.

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

Listing 5-6 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 erec=(EmployeeRecord) in;
         if (!isSsnValid(erec.getEmpRecord().getEmpSsn()))
       {
         // The SSN is not valid.
         throw new Error)”Invalid Social Security Number:”+
              erec.getEmpRecord().getEmpSsm());
       }

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

       //updateEmployee
       //
       public EmployeeRecord updateEmployee(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.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 it has been written, the `ExtSampleClient` class and the other Java source files must be compiled and placed on to your CLASSPATH.

What Do I Do Next?

Refer to *BEA Java Adapter for Mainframe Scenarios* for detailed examples of some of the application models discussed in this guide.

Index