Oracle® Tuxedo

Accessing Mainframe from Java

12*c* Release 2 (12.2.2)

April 2016

ORACLE

Oracle Tuxedo Accessing Mainframe from Java, 12*c* Release 2 (12.2.2)

Contents

Generating a Java Application with the eGen Application Generator

Tuxedo Mainframe Transaction Publisher

Generating a Java Application with the eGen Application Generator

This document includes the following topics:

Generating a Java Application with the eGen Application Generator

Overview

Oracle Tuxedo supports seamless integration of CICS Transaction Gateway (CTG) application running on J2EE application servers and JCA based.

With this feature, Oracle Tuxedo provides a tool to

- parse COBOL copybooks used to describe CICS transactions/programs interfaces
- generate Java bean style classes to populate data

Therefore, users can pass those classes to a CCI (or ECI-wrapped) interface to perform ART-hosted CICS invocations.

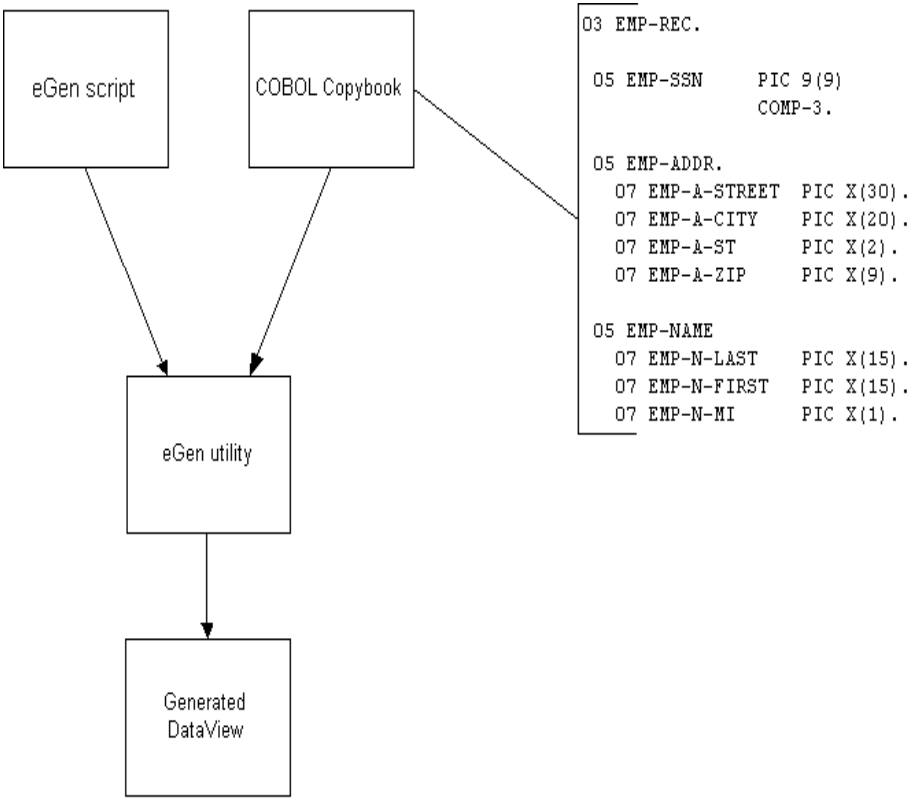
Understanding eGen

eGen Application Generator, also known as the eGen utility, generates Java applications from a COBOL copybook and a user-defined script file.

The eGen utility generates a Java application by processing a script you create, called an eGen script. A Java `DataView` is defined by the first section of the script. This `DataView` is used by the application code to provide data access and conversions, as well as to perform other miscellaneous functions. The actual application code is defined by the second section of the script.

Figure 1 illustrates how the eGen utility works. This illustration shows the eGen script and COBOL copybook file being used as input to the eGen utility, and the output that is generated is the `DataView`.

Figure 1 Understanding the eGen utility



Working with COBOL Copybooks

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 for a CICS application, or in the `WORKING-STORAGE SECTION` of an IMS program. If the CICS or IMS application does not use a copybook file, you will have to create one from the data definition contained in the program source.

Each copybook's contents 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.

Obtaining a COBOL Copybook

The eGen utility must have a COBOL Copybook to use as input. There are two methods you can use to obtain this Copybook:

- Creating a New COBOL Copybook
- Using an Existing COBOL Copybook

Creating a New COBOL Copybook

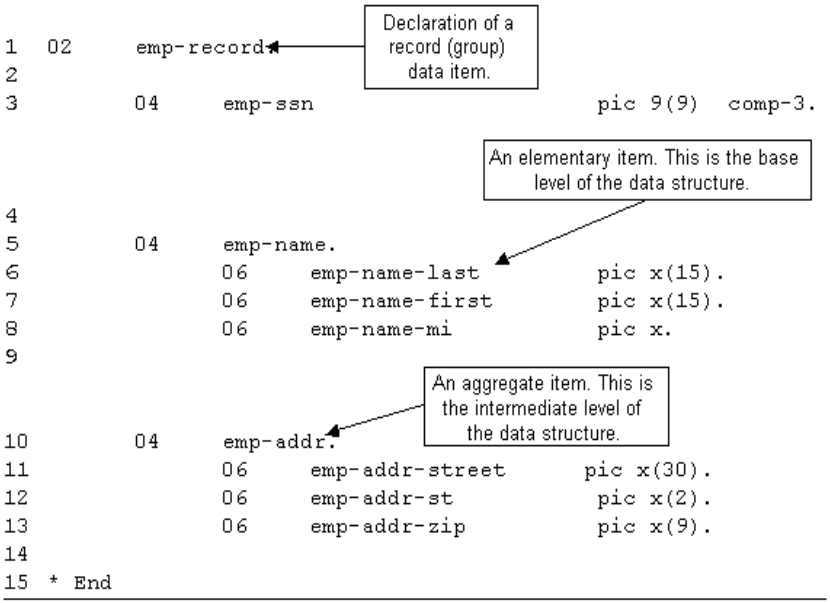
If you are producing a new application on the mainframe or modifying one, then one or more new copybooks may be required. You should keep in mind the COBOL features and data types supported by eGen as you create these copybooks. See "DataView Programming Reference" for more information.

Using an Existing COBOL Copybook

When a mainframe application has an existing DPL or APPC interface, the data for that interface is usually 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 eGen does not support (see "Limitations of the eGen Utility").

See Figure 2 for an example COBOL copybook source file.

Figure 2 Sample emprec.cpy COBOL Copybook



Limitations of the eGen Utility

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 DataView Programming Reference. If the eGen utility is unable to fully support constructs or data types, it:

- Treats them as alphanumeric data types (if reasonable)
- Ignores them
- Reports them as errors

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

Writing an eGen Script

After you have obtained a COBOL Copybook for the mainframe applications, you are ready to write an eGen script. This eGen script and the COBOL copybook that describes your data structure will be processed by the eGen utility to generate a `DataView` which will serve as the basis for your custom Java application.

An eGen script has this section:

- `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. Generating `DataViews` is discussed in detail in the remainder of this section. See "Writing the DataView Section of an eGen Script" for more information.

Writing the DataView Section of an eGen 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 (keywords are in bold).

Listing 1 Sample DataView Section of eGen Script

```
generate view examples.CICS.outbound.gateway.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. `examples.CICS.outbound.gateway.EmployeeRecord` tells the eGen utility to call the DataView file `EmployeeRecord.java`. The package is called `examples.CICS.outbound.gateway`. The `EmployeeRecord` class defined in `EmployeeRecord.java` is a subclass of the DataView class. The phrase from `emprec.cpy` tells the eGen utility to form the **`EmployeeRecord`** DataView file from the COBOL copybook `emprec.cpy`.

Additional generate view statements may be added to an eGen script in order to produce all the DataViews required by your application. Also, 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 2 Sample DataView Section with Codepage Specified

```
…
generate view examples.CICS.outbound.gateway.EmployeeRecord from emprec.cpy
codepage cp500


generate view examples.CICS.outbound.gateway.EmployeeRecord from emprec.cpy
codepage ASCII
```

Listing 3 Sample DataView Section with endian Specified

```
generate view examples.CICS.outbound.gateway.EmployeeRecord from emprec.cpy
endian little
```

Note: By default the `endian` is `big`.

If a jolt client calls a COBOL service in Tuxedo on a Linux X86-64 machine, the jolt client should be compiled with the java code generated by eGen with parameter codepage ASCII and `endian little` in Listing 4.

See "A JOLT Example" for more information.

Listing 4 Sample DataView Section with Parameter Codepage and endian Specified

```
generate view examples.CICS.outbound.gateway.EmployeeRecord from emprec.cpy
codepage ASCII endian little
```

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

Listing 5 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.

Table 1 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`.

Processing eGen Scripts with the eGen Utility

After you have written your eGen script, you must process it to generate the DataView. The same eGen script usually contains the definitions of the DataView, and these definitions are produced with a single processing of the script. However, in this document, the script is 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
- Generating the Java DataView Code

Creating an Environment for Generating and Compiling the Java Code

When you process the eGen scripts and compile 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 this access.

- For the eGen utility:
  - Add `<TUXDIR>\udataobj\egen.jar` to your `CLASSPATH`.
  - Add `<TUXDIR>\bin` to your `PATH`.
- For compilation:
  - Add `<TUXDIR >\udataobj\egen.jar` to your `CLASSPATH`.

– Add the path of your DataView class files to your `CLASSPATH`. You need the 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

For the eGen script shown in Listing 1, the following shell command parses the copybook file (see Figure 2) and generates `EmployeeRecord.java` source file in the current directory:

Listing 6 Sample Copybook Parse Command

```
java com.bea.jam.egen.EgenCobol emprec.egen
```

If no error or warning messages are issued, the copybook is compatible with eGen and the source files are created. Note that no application source files are generated by processing the `emprec.egen` script. This is because there are no application generating commands in this script.

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

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

package examples.CICS.outbound.gateway;

//Imports

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

/**DataView class for EmployeeRecord buffers*/
public final class EmployeeRecord
    extends DataView
{
    ...

    // Code for field "emp-ssn"
    private BigDecimal     m_empSsn;

    public BigDecimal getEmpSsn() {...}

    /** DataView subclass for emp-name Group */
    public final class EmpName3V
            extends DataView
    {
             ...

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

            public void setEmpNameLast(String value) {...}
            public String getEmpNameLast() {...}

            .
            .
            .

    // Code for field "emp-name"
    private EmpName3V  m_empname;
... public Empname3V getEmpname() {...}

}

//End EmployeeRecord.java
```

The package name is defined in the eGen script

The data record is encapsulated in a class that extends the DataView class

Each class member variable corresponds to a field in the data record

Each data field has accessor functions

Each aggregate data field has a corresponding nested inner class that extends the DataView class

Each data field within an aggregate data field has accessor functions

Each COBOL data field name is converted into a Java identifier

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.

Performing Your Own Data Translation

Why Perform Your Own Data Translation?

The automatic data translation provided by DataViews can usually fill your needs. The eGen-generated DataViews relieve your application of the burden of translating data between the mainframe EBCDIC environment and the Java runtime environment. In addition, native mainframe data types that are not supported in Java (such as packed, zoned decimal, etc.) are automatically mapped to appropriate Java data types. However, occasionally you may want to bypass these features and create your own data translation. Following are some advantages of bypassing the eGen/DataView infrastructure:

- Unnecessary data translation may be avoided

  If the data has been acquired in the appropriate format, it can simply be transmitted to the mainframe bypassing the DataView translation overhead.

- Contents of data buffer may be dynamically determined at runtime

In some cases, this may be preferable to a DataView generated from a copybook containing numerous REDEFINES representing various record types.

Simple interfaces are provided for translating data both from and to the mainframe. In addition, a simple `callService()` method is available for making mainframe service requests.

Translating Buffers from Java to Mainframe Representation

Support for creating buffers for input to a mainframe service is provided by the `com.bea.base.io.MainframeWriter` class. This class functions similar to a Java `java.io.DataOutputStream` object. It translates Java data types and all mainframe-native data types. For numeric data types, this translation service provides a conversion from Java native numeric types to those available on the mainframe. For string data types, a translation is performed from UNICODE to EBCDIC by default, although the output codepage used is configurable.

MainframeWriter Public Interface

Listing 7 shows the public methods that `MainframeWriter` class provides.

Listing 7 MainframeWriter Class Public Methods

```
package com.bea.base.io;

public class MainframeWriter

{

        public MainframeWriter();

        public MainframeWriter(String codepage);

        public void setDefaultCodepage(String cp)

        public byte[] toByteArray();

        public void writeRaw(byte[] bytes

                throws IOException;

        public void writeFloat(float value)

                throws IOException;

        public void writeDouble(double value)
```

```
        throws IOException;
public void write(char c)
        throws IOException;
public void writePadded(String s, char padChar, int length)
        throws IOException;
public void write16bit(int value)
        throws IOException;
public void write16bitUnsigned(int value)
        throws IOException;
public void write16bit(long value, int scale)
        throws IOException, ArithmeticException;
public void write16bitUnsigned(long value, int scale)
        throws IOException, ArithmeticException;
public void write32bit(int value)
        throws IOException;
public void write32bitUnsigned(long value)
        throws IOException;
public void write32bit(long value, int scale)
        throws IOException, ArithmeticException;
public void write32bitUnsigned(long value, int scale)
        throws IOException, ArithmeticException;
public void write64bit(long value)
        throws IOException;
public void write64bitUnsigned(long value)
        throws IOException;
public void write64bitBigUnsigned(BigDecimal value)
```

```
            throws IOException;
        public void write64bit(long value, int scale)
            throws IOException, ArithmeticException;
        public void write64bit(BigDecimal value, int scale)
            throws IOException, ArithmeticException;
        public void write64bitUnsigned(long value, int scale)
            throws IOException, ArithmeticException;
        public void write64bitUnsigned(BigDecimal value, int scale)
            throws IOException, ArithmeticException;
        public void writePacked(BigDecimal value, int digits, int precision,
int scale)
            throws ArithmeticException, IOException;
        public void writePackedUnsigned(BigDecimal value, int digits, int
precision, int scale)
            throws ArithmeticException, IOException;
```

Table 2 MainframeWriter Class Public Method Definitions

Method	Description
`MainframeWriter()`	Default constructor. Constructs a `MainframeWriter` using the default code page of `cp037` (EBCDIC).
`MainframeWriter(cp)`	Constructs a `MainframeWriter` using the specified codepage for character field translation.
`setDefaultCodepage(cp)`	Sets the codepage to be used for all future data translations.
`toByteArray()`	Returns the translated buffer constructed by writing data to the `MainframeWriter` class as a byte array.
`writeRaw(bytes)`	Writes a raw byte array to the output buffer.

Table 2 MainframeWriter Class Public Method Definitions

Method	Description
`writeFloat(num)`	Converts a floating point number from IEEE Java float data type to IBM 4 byte floating point format. The equivalent COBOL picture clause is `PIC S9V9 COMP-1`.
`writeDouble(num)`	Converts a floating point number from IEEE Java double data type to IBM 8 byte floating point format. The equivalent COBOL picture clause is `PIC S9V9 COMP-2`.
`write(c)`	Translates and writes a single character to output buffer. The equivalent COBOL picture clause is `PIC X`.
`writePadded(str, pad, len)`	Translate and write a string to a fixed length character field. The passed pad character is used if the length of the passed string is less than `len`. If the length of the passed string is greater than `len`, it will be truncated to `len` characters. The equivalent COBOL picture clause is `PIC X(len)`.
`write16bit(num)`	Writes a signed 16 bit binary integer to output buffer. The equivalent COBOL picture clause is `PIC S9(4) COMP`.
`write16bitUnsigned(num)`	Writes an unsigned 16 bit binary integer to output buffer. The equivalent COBOL picture clause is `PIC 9(4) COMP`.
`write16bit(num, scale)`	Writes a signed 16 bit integer to the output buffer after moving the implied decimal point left by `scale` digits. For example, the call `write16bit(100, 1)` would result in the value 10 being written. The equivalent COBOL picture clause is `PIC S9(4) COMP`.
`write16bitUnsigned(num, scale)`	Writes an unsigned 16 bit integer to the output buffer after moving the implied decimal point left by `scale` digits. For example, the call `write16bitUnsigned(100, 1)` would result in the value 10 being written. The equivalent COBOL picture clause is `PIC 9(4) COMP`.
`write32bit(num)`	Writes a signed 32 bit binary integer to output buffer. The equivalent COBOL picture clause is `PIC S9(8) COMP`.
`write32bitUnsigned(num)`	Writes an unsigned 32 bit binary integer to output buffer. The equivalent COBOL picture clause is `PIC 9(8) COMP`.

Table 2 MainframeWriter Class Public Method Definitions

Method	Description
`write32bit(num, scale)`	Writes a signed 32 bit integer to the output buffer after moving the implied decimal point left by scale digits. For example, the call `write32bit(100L, 1)` would result in the value 10 being written. The equivalent COBOL picture clause is `PIC S9(8) COMP`.
`write32bitUnsigned(num, scale)`	Writes an unsigned 32 bit integer to the output buffer after moving the implied decimal point left by `scale` digits. For example, the call `write32bitUnsigned(100L, 1)` would result in the value 10 being written. The equivalent COBOL picture clause is `PIC 9(8) COMP`.
`write64bit(num)`	Writes a signed 64 bit binary integer to output buffer. The equivalent COBOL picture clause is `PIC S9(15) COMP`.
`write64bitUnsigned(num)`	Writes an unsigned 64 bit binary integer to output buffer. The equivalent COBOL picture clause is `PIC 9(15) COMP`.
`write64bit(num, scale)`	Writes a signed 64 bit integer to the output buffer after moving the implied decimal point left by `scale` digits. For example, the call `write64bit(100L, 1)` would result in the value 10 being written. The equivalent COBOL picture clause is `PIC S9(15) COMP`.
`write64bitUnsigned(num, scale)`	Writes an unsigned 64 bit integer to the output buffer after moving the implied decimal point left by `scale` digits. For example, the call `write64bitUnsigned(100L, 1)` would result in the value 10 being written. The equivalent COBOL picture clause is `PIC 9(15) COMP`.
`writePacked(num, digits, rec, scale)`	Writes a decimal number as an IBM signed packed data type with `digits` decimal digits total and `prec` digits to the right of the decimal point. Prior to conversion, the number is scaled to the left `scale` digits. The equivalent COBOL picture clause is `PIC S9(digits-prec)V9(prec) COMP-3`.
`writePackedUnsigned(num, digits, prec, scale)`	Writes a decimal number as an IBM unsigned packed data type with `digits` decimal digits total and `prec` digits to the right of the decimal point. Prior to conversion the number is scaled to the left `scale` digits. The equivalent COBOL picture clause is `PIC 9(digits-prec)V9(prec) COMP-3`.

Using MainframeWriter to Create Data Buffers

As an example of using the `MainframeWriter` class to create a mainframe data buffer, assume we have a mainframe service which accepts the data record shown as below.

Listing 8 Data Record

```
01 INPUT-DATA-REC.
       05 FIRST-NAME        PIC X(10).
       05 LAST-NAME         PIC X(10).
       05 AGE               PIC S9(4) COMP.
       05 HOURLY-RATE       PIC S9(3)V9(2) COMP-3.
```

Listing 9 shows a Java test program that creates a buffer matching this record layout using `MainframeWriter` translation class:

Listing 9 Java Test Program

```
import java.math.BigDecimal;
import com.bea.base.io.MainframeWriter;
public class MakeBuffer
{
       public static void main(String[] args) throws Exception
       {
              MainframeWriter mf = new MainframeWriter();
              mf.writePadded("Edgar", ' ', 10);//first name
              mf.writePadded("Jones", ' ', 10);//last name
              mf.write16bit(22);//age
              mf.writePacked(new BigDecimal(22.50), 5, 2, 0);//hourly rate
```

```
                byte[] buffer = mf.toByteArray();
                System.out.println(getHexString(buffer));
        }
        private static String getHexString(byte[] buffer)
        {
                StringBuffer hexStr = new StringBuffer(buffer.length * 2);
                for (int i = 0; i < buffer.length; ++i)
                {
                        int n = buffer[i] & 0xff;
                        hexStr.append(hex[n >> 4]);
                        hexStr.append(hex[n & 0x0f]);
                }
                return(hexStr.toString());
        }
        private static char[] hex =
        {
                '0', '1', '2', '3', '4', '5', '6', '7',
                '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
        };
}
```

The output of running this sample program is:

```
C5848781994040404040D1969585A240404040400001602250C
```

This buffer breaks down as follows:

```
FIRST-NAME    C5848781994040404040"Edgar" + 5 spaces in EBCDIC
LAST-NAME     D1969585A24040404040"Jones" + 5 spaces in EBCDIC
AGE           001622 as 16 bit integer
```

```
HOURLY-RATE   02250C22.50 positive packed number
                            (decimal point is assumed)
```

Translating Buffers from Mainframe Format to Java

Support for translating data received from the mainframe to Java data types is provided by the `com.bea.base.io.MainframeReader` class. This class operates in a manner similar to a Java `jam.io.DataInputStream`, and performs translations from mainframe data types to equivalent types usable by a Java program. Like the MainframeWriter class, the codepage used for string translations may be configured and defaults to EBCDIC.

MainframeReader Public Interface

Listing 10 shows the public methods that `MainframeReader` class provides.

Listing 10 MainframeReader Class Public Methods

```
package com.bea.base.io;

public class MainframeReader

{

        public MainframeReader(byte[] buffer);

        public MainframeReader(byte[] buffer, String codepage);

        public void setDefaultCodepage(String cp);

        public byte[] readRaw(int count) throws IOException;

        public float readFloat() throws IOException;

        public double readDouble() throws IOException;

        public char readChar() throws IOException;

        public String readPadded(char padChar, int length)

                throws IOException;

        public short read16bit() throws IOException;

        public int read16bitUnsigned() throws IOException;

        public long read16bit(int scale) throws IOException;
```

```
        public int read32bit() throws IOException;
        public long read32bit(int scale)
                throws IOException;
        public long read32bitUnsigned() throws IOException;
        public long read32bitUnsigned(int scale) throws IOException;
        public long read64bit() throws IOException;
        public long read64bitUnsigned()
                throws IOException;
        public long read64bit(int scale)
                throws IOException;
        public BigDecimal read64bitBigUnsigned()
                throws IOException;
        public BigDecimal read64bitBig(int scale)
                throws IOException
       public BigDecimal readPackedUnsigned(int digits, int precision, int
scale)
                throws ArithmeticException, IOException;
        public BigDecimal readPacked(int digits, int precision, int scale)
                throws ArithmeticException, IOException;
}
```

Following are the definitions of these methods:

Table 3 MainframeReader Class Public Method Definitions

Method	Description
`MainframeReader(buffer)`	Constructs a `MainframeReader` for the passed buffer using the default code page of `cp037` (EBCDIC).
`MainframeReader(buffer, cp)`	Constructs a `MainframeReader` for the passed buffer using the specified codepage for character field translation.
`setDefaultCodepage(cp)`	Sets the codepage to be used for all future character translations.
`readRaw(count)`	Reads count characters from the buffer without any translation and returns them as a byte array.
`readFloat()`	Reads a four byte IBM floating point number and returns it as a Java float data type.
`readDouble()`	Reads an eight byte IBM floating point number and returns it as a Java double data type.
`readChar()`	Reads and translates a single character.
`readPadded(pad, len)`	Reads and translates a fixed length character field and returns it as a Java String. The length of the field is passed as `len` and the field pad character is passed as `pad`. Trailing instances of the `pad` character are removed before the data is returned.
`read16bit()`	Reads a 16 bit binary integer and returns it as a Java short.
`read16bitUnsigned()`	Reads an unsigned 16 bit integer and returns it as a Java int.
`read16bit(scale)`	Reads a 16 bit binary integer and scales the value by 10^scale. For example, if value 10 is read via `read16bit(1)`, the returned value would be 100.
`read32bit()`	Reads a 32 bit binary integer and returns it as a Java int.
`read32bit(scale)`	Reads a 32 bit binary integer and scales the value by 10^scale. For example, if value `10` is read via `read32bit(1)`, the returned value would be 100.
`read32bitUnsigned()`	Reads an unsigned 32 bit integer and returns it as a Java long.

Table 3 MainframeReader Class Public Method Definitions

Method	Description
`read32bitUnsigned(scale)`	Reads an unsigned 32 bit binary integer and scales the value by 10^scale. For example, if value `10` is read via `read32bit(1)`, the returned value would be 100.
`read64bit()`	Reads a 64 bit binary integer and returns it as a Java long.
`read64bitUnsigned()`	Reads an unsigned 64 bit integer and returns it as a Java long.
`read64bitUnsigned(scale)`	Reads an unsigned 64 bit binary integer and scales the value by 10^scale. For example, if value `10` is read via `read32bit(1)`, the returned value would be 100.
`read64bitBigUnsigned()`	Reads an unsigned 64 bit integer and returns it as a Java `BigDecimal`.
`read64bitBig(scale)`	Reads a signed 64 bit integer and scales the value by 10^scale. The value is returned as a Java `BigDecimal`.
`readPackedUnsigned(digits, prec, scale)`	Reads an unsigned packed number consisting of digits numeric digits with `prec` digits to the right of the decimal. The value is scaled by 10^scale and is returned as a Java `BigDecimal`.
`readPacked(digits, prec, scale)`	Reads a signed packed number consisting of `digits` numeric digits with `prec` digits to the right of the decimal. The value is scaled by 10^scale and is returned as a Java `BigDecimal`.

Using MainframeReader to Translate Data Buffers

As an example of using the `MainframeReader`, class following is a program that translates and displays the fields in the mainframe buffer created above. Our input buffer consists of the binary data:

```
C5848781994040404040D1969585A24040404040001602250C
```

Listing 11 shows the sample program used to process this buffer.

Listing 11 Sample Program

```
import java.math.BigDecimal;

import com.bea.base.io.MainframeReader;
```

```
public class ShowBuffer
{
       public static void main(String[] args) throws Exception
       {
              String data
="C5848781994040404040D1969585A2404040404040001602250C";
              byte[] buffer = buildBinary(data);
              MainframeReader mf = new MainframeReader(buffer);
              System.out.println(" First Name: " + mf.readPadded(' ', 10));
              System.out.println(" Last Name: " + mf.readPadded(' ', 10));
              System.out.println("Age: " + mf.read16bit());
              System.out.println("Hourly Rate: " + mf.readPacked(5, 2, 0));
       }
       private static byte[] buildBinary(String data)
       {
              byte[] buffer = new byte[data.length() / 2];
              for (int i = 0; i < buffer.length; ++i)
              {
              int msb = hex.indexOf(data.charAt(i * 2));
              int lsb = hex.indexOf(data.charAt(i * 2 + 1));
               buffer[i] = (byte) (msb << 4 | lsb);
              }
              return(buffer);
       }
       private static final String hex = "0123456789ABCDEF";
}
```

When running, the program produces the following output:

```
First Name: Edgar

Last Name: Jones

Age: 22
```

DataView Programming Reference

This section provides the rules that allow you to identify what form a generated Java class takes from a given COBOL copybook processed by the eGen Application Generator (eGen utility). An understanding of the rules facilitates a programmer's ability to correctly code any custom programs that make use of the generated classes.

The eGen utility maps a COBOL copybook into a Java class. The COBOL copybook contains a data record description. The eGen utility derives the generated Java class from the `com.bea.dmd.dataview.DataView` class (later referred to as `DataView`).

This section discusses data mapping rules in the following topics:

- Field Name Mapping Rules
- Field Type Mappings
- Group Field Accessors
- Elementary Field Accessors
- Array Field Accessors
- Fields with REDEFINES Clauses
- COBOL Data Types
- Other Access Methods for Generated DataView Classes
- Known Limitations of eGen working with COBOL Copybooks

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

Field Name Mapping Rules

When you process a COBOL copybook containing field names, they are mapped to Java names by the eGen utility. All alphabetic characters are mapped to lower case, except in the following two cases.

All dashes are removed and the character following the dash is mapped to upper case.

When a prefix is added to the name (as when creating a field accessor function name), the first character of the base name is mapped to upper case.

Table 4 lists some mapping examples.

Table 4 Example Field Name Mapping from COBOL to Java and Accessor

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

Field Type Mappings

When you process a COBOL copybook, the data types of fields are mapped to Java data types. The mapping is performed by the eGen utility according to the following rules:

1. Groups map to `DataView` subclasses.
2. All alphanumeric fields are mapped to type `String`.
3. All edited numeric fields are mapped to type `String`.
4. All `SIGN SEPARATE`, `BLANK WHEN ZERO` or `JUSTIFIED RIGHT` fields are mapped to type String.
5. `SIGN IS LEADING` is not supported.
6. The types `COMP-1`, `COMP-2`, `COMP-5`, `COMP-X`, and `PROCEDURE-POINTER` fields are not supported (an error message is generated).
7. All `INDEX` fields are mapped to Java type int.
8. `POINTER` maps to Java type `int`.

9. All numeric fields with any digits to the right of the decimal point are mapped to type `BigDecimal`.
10. All `COMP-3` (packed) fields are mapped to type `BigDecimal`.
11. All other numeric fields are mapped as shown in Table 5.

Table 5 Numeric Field Mapping

Number of Digits	Java Type
<= 4	short
> 4 and <= 9	int
> 9 and <= 18	long
> 18	BigDecimal

Group Field Accessors

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

For example, the following copybook:

Listing 12 Sample Copybook

```
10 MY-RECORD.
       20  MY-GRP.
              30  ALNUM-FIELD             PIC X(20).
Produces code similar to the following:
 public MyGrp2V getMyGrp();
public static class MyGrp2V extends DataView
{
              // Class definition
}
```

Elementary Field Accessors

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

These accessors have the following forms:

```
public void setFieldName(FieldType value);

public FieldType getFieldName();
```

Where:

`FieldType` is described in the Field Type Mappings section.

`FieldName` is described in the Field Name Mapping Rules section.

For example, the following copybook:

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

Produces the accessors:

```
public void setNumericField(int value);

public int getNumericField();

public void setAlnumField(String value);

public String getAlnumField();
```

Array Field Accessors

Array fields are handled according to the field accessor rules described in Group Field Accessors and Elementary Field Accessors, with the addition that each accessor takes an additional `int` argument that specifies which array entry is to be accessed, for example:

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

public FieldType     getFieldName(int index);
```

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

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

Fields with REDEFINES Clauses

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

For example the copybook:

Listing 13 Sample Copybook

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

Produces Java code similar to the following:

```
private byte[] m_redef23;
private DataView m_redef23DV;
public InputDataV   getInputData();
public String getOutputData();
public void   setOutputData(String value);
public static class InputDataV extends DataView
```

```
{
// Class definition.
}
```

COBOL Data Types

This section summarizes the COBOL data types supported by Tuxedo. Table 6 lists the COBOL data item definitions recognized by the eGen utility. Table 7 lists the syntactical features and data types recognized by the eGen utility. If a COBOL feature is unsupported and it is not listed as ignored in the table, an error message is generated.

Table 6 Major COBOL Features

COBOL Feature	Support
IDENTIFICATION DIVISION	Unsupported
ENVIRONMENT DIVISION	Unsupported
DATA DIVISION	Partially Supported
WORKING-STORAGE SECTION	Partially Supported
Data record definition	Supported
PROCEDURE DIVISION	Unsupported
COPY	Unsupported
COPY REPLACING	Unsupported
EJECT, SKIP1, SKIP2, SKIP3	Supported

Table 7 COBOL Data Types

COBOL Type	Java Type
COMP, COMP-4, BINARY (integer)	Short/Int/Long
COMP, COMP-4, BINARY (fixed)	BigDecimal
COMP-3, PACKED-DECIMAL	BigDecimal

Table 7 COBOL Data Types

COBOL Type	Java Type
COMP-5	Unsupported
COMP-X	Unsupported
DISPLAY numeric (zoned)	BigDecimal
BLANK WHEN ZERO (zoned)	String
SIGN IS LEADING (zoned)	Unsupported
SIGN IS LEADING SEPARATE (zoned)	String
SIGN IS TRAILING (zoned)	String
SIGN IS TRAILING SEPARATE (zoned)	String
edited numeric	String
COMP-1, COMP-2 (float)	Unsupported
edited float numeric	String
DISPLAY (alphanumeric)	String
edited alphanumeric	String
INDEX	Int
POINTER	Int
PROCEDURE-POINTER	Unsupported
JUSTIFIED RIGHT	Unsupported (ignored)
SYNCHRONIZED	Unsupported (ignored)
REDEFINES	Supported
66 RENAMES	Unsupported
66 RENAMES THRU	Unsupported
77 level	Supported
88 level (condition)	Unsupported (ignored)

Table 7 COBOL Data Types

COBOL Type	Java Type
group record	Inner Class
OCCURS (fixed array)	Array
OCCURS DEPENDING (variable-length array)	Array
OCCURS INDEXED BY	Unsupported (ignored)
OCCURS KEY IS	Unsupported (ignored)

Other Access Methods for Generated DataView Classes

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

- Mainframe Access to DataView Classes
- XML Access to DataView Classes
- Hashtable Access to DataView Classes

Mainframe Access to DataView Classes

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

Mainframe format data may be extracted from a DataView class through the use of the `MainframeWriter` class. Listing 14 shows a sample of code that may be used to perform the extraction.

Listing 14 Sample Code for Extracting Mainframe Format Data from a DataView Class

```
import com.bea.base.io.MainframeWriter;

import com.bea.dmd.dataview.DataView;

...

/**
```

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

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

If you want to override the codepage provided when the DataView was generated, you may provide another codepage as a `String` argument to the `MainframeWriter` constructor, as shown in the comment in Listing 15.

Loading mainframe data into a DataView is a similar process, in this case requiring the use of the `MainframeReader` class. Listing 15 shows a sample of code that may be used to perform the load.

Listing 15 Sample Code for Loading Mainframe Data into a DataView Class

```
import com.bea.base.io.MainframeReader;
import com.bea.dmd.dataview.DataView;
...
/**
* Put a byte[] containing mainframe format data into a DataView.
 */
MyDataViewputMainframeData(byte[] buffer)
{
      MainframeReader mr = new MainframeReader(buffer);
      // To override the DataView's codepage, change the above
      // constructor call to something like:
      // …new MainframeReader("cp1234", buffer);
    .
    .
    .
MyDataView dv;
    .
    .
    .
try
       {
       // Construct a new DataView with the mainframe data.
              dv = new MyDataView(mr);
              // Or, to load a pre-existing DataView with mainframe data.
       // dv.mainframeLoad(mr);
       }
```

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

XML Access to DataView Classes

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

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

The following list shows an example of the code used to load XML data into a DataView.

Listing 16 Sample Code for Loading XML Data into a DataView

```
import com.bea.dmd.dataview.DataView;
import com.bea.dmd.dataview.XmlLoader;
...
        void loadXmlData(String xml, DataView dv)
        {
                XmlLoader xl = new XmlLoader();
                try
                {
```

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

The following list shows an example of the code used to unload a `DataView` into XML.

Listing 17 Sample Code for Unloading a DataView into XML

```
import com.bea.dmd.dataview.DataView;
import com.bea.dmd.dataview.XmlUnloader;
...
        String unloadXmlData(DataView dv)
        {
                XmlUnloader xu = new XmlUnloader();
                try
                {
                        String xml = xu.unload(dv);
                        return xml;
                }
                catch (Exception e)
                {
```

```
                    // Some conversion error occurred.
                }
                return null;
        }
```

Hashtable Access to DataView Classes

Oracle Tuxedo also provides facilities to load and unload DataView objects using Hashtable objects. Hashtable objects are most often used to move data from one DataView to another similar DataView.

When DataView fields are moved into Hashtables, each field is given a key that is a string reflecting the location of the field within the original copybook data structure.

Listing 18 shows a sample of a COBOL copybook.

Listing 18 Sample emprec.cpy COBOL Copybook

```
1       *-------------------------------------------------------
2       * emprec.cpy
3       *       An employee record.
4       *-------------------------------------------------------
5
6       02      emp-record.
7
8               04      emp-ssn                     pic 9(9)  comp-3.
9
10              04      emp-name.
11                      06      emp-name-last       pic x(15).
12                      06      emp-name-first      pic x(15).
13                      06      emp-name-mi         pic x.
```

```
14
15            04     emp-addr.
16                   06     emp-addr-street      pic x(30).
17                   06     emp-addr-st          pic x(2).
18                   06     emp-addr-zip         pic x(9).
19
20     * End
```

The fields for the COBOL copybook in Listing 18 are stored into a Hashtable as shown in Table 8.

Table 8 COBOL Copybook Hashtable

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

Code for Unloading and Loading Hashtables

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

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

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

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

Rules for Unloading and Loading Hashtables

The basic rules of Hashtable unloading are:

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

The basic rules of Hashtable loading are:

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

Name Translator Interface Facility

A name translator interface facility is available to provide `Hashtable` name mappings. Both `HashtableLoader` and `HashtableUnloader` provide a constructor that accepts an argument of type `com.bea.dmd.dataview.NameTranslator`. Table 9 lists the descriptions of the public interface methods that must be implemented.

Table 9 Name Translator Interface

Method	Description
`translate(String input)`	This method received a `String` object as an input parameter and returns a `String` object.

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

Following are some useful implementations that are included in the `egen.jar`:

Table 10 Name Translator Interface

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

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

Known Limitations of eGen working with COBOL Copybooks

Following are some of the known limitations of this version of eGen.

- Continuation lines are not recognized in COBOL copybooks. This is only a problem for long character literals occurring within `VALUES` clauses. Comment out the relevant clause to fix the problem.
- COBOL copybooks with array (table) data items having an `OCCURS DEPENDING ON` clause must be structured so that the depending-on counter data item is not contained within the same group data item as the one containing the array.
- `USAGE` clauses on group data items in COBOL copybooks are not properly propagated to their subordinated member data items.

Program Development

Program development will be accomplished according to program snippet listed in Listing 19 and according to class naming rules outlined here, although this can be adjusted depending on customer requirements.

Listing 19 Program Snippet

```
try
     {
```

```
 InitialContext context = new InitialContext();

ECIConnectionSpec connSpec = new ECIConnectionSpec();
connSpec.setUserName("TESOP01");
connSpec.setPassword("");
Connection connection = connectionFactory.getConnection(connSpec);

 Interaction interaction = connection.createInteraction();

 // Create inputBean
         K294Bean inRec = new K294Bean();
         inRec.setI__Entete__TranId("K294");
         inRec.setI__Entete__Vers("0101");
         inRec.setI__Entete__Statut("99");
         inRec.setI__Entete__Nb__Enreg((short)40);
         inRec.setI__Entete__User("TESOP01");
         inRec.setI__Entete__Date("2012-01-16");

// Data
         inRec.setI__restea__nupy(1);
         inRec.setI__restea__cdea(2);
         inRec.setI__restea__cdea1(1);

         K294Bean outRec = new K294Bean();

// Create InteractionSpec
  InteractionSpec interactionSpec = new ECIInteractionSpec();
```

```
        ((ECIInteractionSpec)interactionSpec).setFunctionName("COMPT294");
        ((ECIInteractionSpec)interactionSpec).setTranName("K294");
        ((ECIInteractionSpec)interactionSpec).setCommareaLength(7132);

((ECIInteractionSpec)interactionSpec).setInteractionVerb(ECIInteractionSpe
c.SYNC_SEND_RECEIVE);

        // execute transaction
        interaction.execute((ECIInteractionSpec)interactionSpec, inRec,
outRec);

        // Close all
                interaction.close();
                connection.close();

       // List Data
           K294bean_output__message_t__o__data__data  data[] =
outRec.getT__o__data__data();

      // Load List
      for (int i=0; i<data.length;i++)
      {
               if (data[i].getT__o__data__data__o__restea__cdea()!=0)
                 {
                   out.println(data[i]);
                 }
      }
  }
```

```
catch (Exception e)
      {
        System.out.println("Error : " + e.getMessage());
        e.printStackTrace();
      }
```

Important Areas

The following listings show the important areas for program development. Field name mappings may vary.

- Listing 20, “Setup Connection
- Listing 21, “Input Bean Usage
- Listing 22, “Service Invocation
- Listing 23, “Output Bean Usage

Listing 20 Setup Connection

```
ECIConnectionSpec connSpec = new ECIConnectionSpec();
       connSpec.setUserName("TESOP01");
       connSpec.setPassword("");
       Connection connection = connectionFactory.getConnection(connSpec);

        Interaction interaction = connection.createInteraction();
  // Create InteractionSpec
        InteractionSpec interactionSpec = new ECIInteractionSpec();
        ((ECIInteractionSpec)interactionSpec).setFunctionName("COMPT294");
        ((ECIInteractionSpec)interactionSpec).setTranName("K294");
        ((ECIInteractionSpec)interactionSpec).setCommareaLength(7132);
```

```
((ECIInteractionSpec)interactionSpec).setInteractionVerb(ECIInteractionSpe
c.SYNC_SEND_RECEIVE);
```

Listing 21 Input Bean Usage

```
// Create inputBean
                K294Bean inRec = new K294Bean();
                inRec.getDfhcommarea().
                    getInputMessage().
                    getIEntete().setIEnteteTranId("K294");
                inRec.getDfhcommarea().
                    getInputMessage().
                    getIEntete().setIEnteteVers("0101");
                inRec.getDfhcommarea().
                    getInputMessage().
                    getIEntete().setIEnteteStatut("99");
                inRec.getDfhcommarea().
                    getInputMessage().
                    getIEntete().setIEnteteNbEnreg((short)40);
// reserve outputBean
          K294Bean outRec = new K294Bean();
```

Listing 22 Service Invocation

```
// execute transaction
        interaction.execute((ECIInteractionSpec)interactionSpec, inRec,
outRec);
```

Listing 23 Output Bean Usage

```
K294bean_output__message_t__o__data__data  data[] =
outRec.getDfhcommarea().getOutputMessage().getTODataData();
```

A JOLT Example

Below is the COBOL copybook `emprec.cpy`.

Listing 24 Sample COBOL copybook emprec.cpy

```
01  emp-record.
           04 emp-ssn                    pic 9(9) comp-3.
           04 emp-name.
                   06 emp-name-last      pic x(15).
                   06 emp-name-first     pic x(15).
                   06 emp-name-mi        pic x.

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

- On Linux machine, you could define eGen script emprec.egen as below.

  ```
  generate view  test.EmployeeRecord from emprec.cpy codepage ASCII endian
  little
  ```

- Next, you could process eGen script `emprec.egen` as below, and then java file `EmployeeRecord.java` is generated.

  ```
  java com.bea.jam.egen.EgenCobol emprec.egen
  ```

- Next, after compiling `EmployeeRecord.java`, you will get four java class files.

  `EmployeeRecord$EmpRecord1V$EmpAddr7V.class`

  `EmployeeRecord$EmpRecord1V$EmpName3V.class`

  `EmployeeRecord$EmpRecord1V.class`

  `EmployeeRecord.class`

- Next, you can write jolt client java code with `EmployeeRecord.java`. See below for a simple example.

Listing 25 Sample Java Code

```
import bea.jolt.*;
import java.math.BigDecimal;
import com.bea.base.io.MainframeWriter;
import com.bea.base.io.MainframeReader;
import java.io.IOException;
import com.bea.sna.jcrmgw.snaException;
import test.*;



public class jc {
       public static void main ( String[] args ) {
              JoltSession                    jses;

              try {
                     JoltSessionAttributes   jattr;
                     JoltRemoteService       toupper,addsvc;
                     JoltTransaction         trans;
                     String name=null;
```

```
                    String pass=null;
                    String apass=null;
                    String urole="myapp";
                    String outstr,addr;
                    test.EmployeeRecord egenclass;
                    BigDecimal   value;

                    jattr = new JoltSessionAttributes();

                    //jattr.setString(jattr.APPADDRESS, "//lcdux2:5555");
                    jattr.setString(jattr.APPADDRESS, "//"+args[0]);
                    jattr.setInt(jattr.IDLETIMEOUT, 300);
                    jattr.setString(jattr.TRUSTSTORE,
"wallet/trust.jks");
                    jattr.setString(jattr.TSPASSPHRASE, "abcd1234");
                    jses = new JoltSession(jattr, name, urole, pass,
apass);

                    String testString = new String("john");

                    egenclass = new test.EmployeeRecord();
                    value = new BigDecimal("123456789");
                    egenclass.getEmpRecord().setEmpSsn(value);

egenclass.getEmpRecord().getEmpName().setEmpNameFirst(testString);
                    byte[]  inputBuffer = egenclass.toByteArray(new
MainframeWriter());
```

```
                    toupper = new JoltRemoteService("CSIMPSRV", jses);
                    toupper.setBytes("DATAFLOW", inputBuffer,
inputBuffer.length);
                    toupper.call(null);
                    byte[] rawResult= null;
                    rawResult =  toupper.getBytesDef("DATAFLOW", null);
                    test.EmployeeRecord result = new
test.EmployeeRecord(new MainframeReader(rawResult));
                    value = result.getEmpRecord().getEmpSsn();
                    System.out.println("after call  emp-ssn is " +
value.toString() );
                    jses.endSession();
                    System.exit(0);
                       } // end of try block
               catch (SessionException e )
                       {
                       System.err.println(e);
                       System.exit(1);
                       } // catch of try block
               catch (IOException ioe )
                       {
                       System.err.println(ioe);
                       System.exit(1);
                       }
               } // of main
       } // public class jc
```

- One Tuxedo server site, you can write a Tuxedo COBOL server that uses the same copybook `emprec.cpy`. See below for a simple example.

Listing 26 Sample on Tuxedo Server Site

```
*
         IDENTIFICATION DIVISION.
         PROGRAM-ID. CSIMPSRV.
         AUTHOR. TUXEDO DEVELOPMENT.
         ENVIRONMENT DIVISION.
         CONFIGURATION SECTION.
       *
         SPECIAL-NAMES. CONSOLE IS CRT.
       *
         DATA DIVISION.
         WORKING-STORAGE SECTION.
             copy 'emprec'.
       ******************************************************
       * Tuxedo definitions
       ******************************************************
         01 TPSVCRET-REC.
         COPY TPSVCRET.
       *
         01 TPTYPE-REC.
         COPY TPTYPE.
       *
         01 TPSTATUS-REC.
         COPY TPSTATUS.
```

```
*
  01 TPSVCDEF-REC.
  COPY TPSVCDEF.
*******************************************************
* Log messages definitions
*******************************************************
  01 LOGMSG.
     05 FILLER         PIC X(10) VALUE "CSIMPSRV :".
     05 LOGMSG-TEXT    PIC X(50).
  01 LOGMSG-LEN              PIC S9(9) COMP-5.
*******************************************************
* User defined data records
*******************************************************
   01 RECV-STRING             PIC X(100).
   01 SEND-STRING             PIC X(100).
*
   LINKAGE SECTION.
*
   PROCEDURE DIVISION.
*
   START-FUNDUPSR.
   MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
   MOVE "Started" TO LOGMSG-TEXT.
   PERFORM DO-USERLOG.

*******************************************************
* Get the data that was sent by the client
```

```
*****************************************************
   MOVE LENGTH OF RECV-STRING TO LEN.
   CALL "TPSVCSTART" USING TPSVCDEF-REC
                TPTYPE-REC
                                        emp-record
                                        TPSTATUS-REC.

   IF NOT TPOK
          MOVE "TPSVCSTART Failed" TO LOGMSG-TEXT
          PERFORM DO-USERLOG
          PERFORM EXIT-PROGRAM
           END-IF.

   IF TPTRUNCATE
          MOVE "Data was truncated" TO LOGMSG-TEXT
          PERFORM DO-USERLOG
          PERFORM EXIT-PROGRAM
   END-IF.

   MOVE emp-ssn TO LOGMSG-TEXT.
   PERFORM DO-USERLOG.

   MOVE 987654321 to emp-ssn.

   MOVE emp-name-first TO LOGMSG-TEXT.
   PERFORM DO-USERLOG.
```

```
      MOVE "Success" TO LOGMSG-TEXT.
      PERFORM DO-USERLOG.
      SET TPSUCCESS TO TRUE.
      COPY TPRETURN REPLACING
             DATA-REC BY emp-record.

   ******************************************************
   * Write out a log err messages
   ******************************************************
    DO-USERLOG.
      CALL "USERLOG" USING LOGMSG
             LOGMSG-LEN
             TPSTATUS-REC.
   ******************************************************
   * EXIT PROGRAM
   ******************************************************
    EXIT-PROGRAM.
      MOVE "Failed" TO LOGMSG-TEXT.
      PERFORM DO-USERLOG.
      SET TPFAIL TO TRUE.
      COPY TPRETURN REPLACING
             DATA-REC BY emp-record.
```

- Next, set the correct COBOL compile environment. For example:

```
export COBDIR=/opt/cobol-it-64
export TM_COBOLIT_VERSION=3.7
```

```
export COBOLIT_LICENSE=$COBDIR/citlicense.xml
export PATH=$COBDIR/bin:$PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$COBDIR/lib
export COBCPY=$TUXDIR/cobinclude
```

- Last, the COBOL server is compiled as below.

```
buildserver -C -o CSIMPSRV -f CSIMPSRV.cbl -f TPSVRINIT.cbl -s CSIMPSRV
```

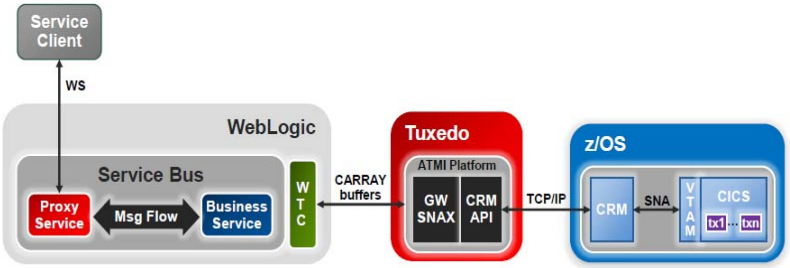
Tuxedo Mainframe Transaction Publisher

This document includes the following topics:

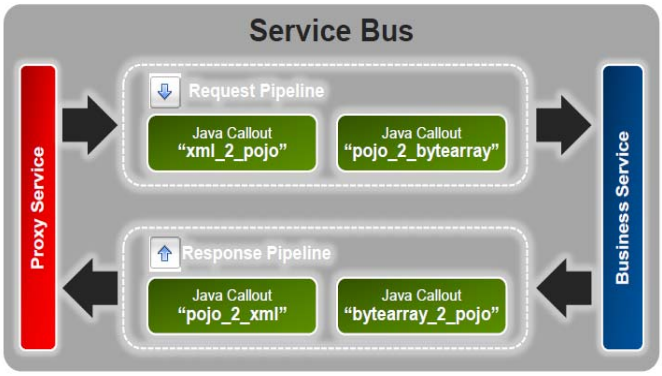
Overview

Tuxedo Mainframe Transaction Publisher simplifies the process of exposing mainframe transaction in Oracle Service Bus (OSB) by providing a graphical user interface.

Let us consider this scenario, where users want to expose their mainframe transaction in OSB. The proxy service uses WSDL and the business service uses WTC.



The tool generates POJO code based on the input COBOL copybook. These generated codes can be used by users to access mainframe transaction.

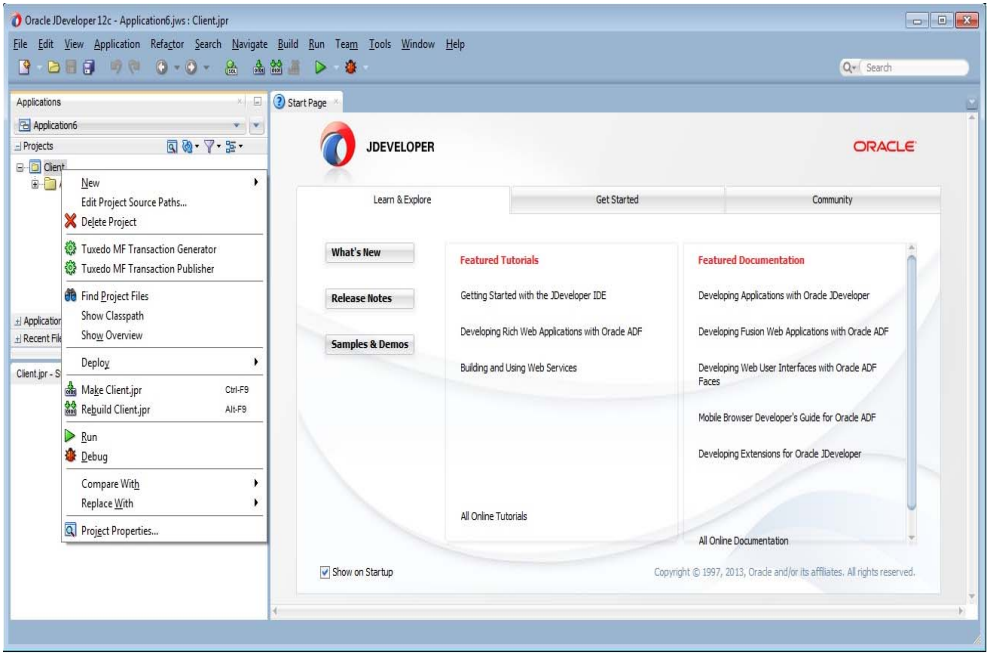


Using Tuxedo Mainframe Transaction Publisher

Tuxedo Mainframe Transaction Publisher includes two parts: Generator and Publisher. They are implemented as JDeveloper extensions and reside in a single JAR file.

- Tuxedo Mainframe Transaction Generator
- Tuxedo Mainframe Transaction Publisher

Tuxedo Mainframe Transaction Publisher is a project based tool. Users select the project and right click to bring up context menu.



Note: Users install this extension using JDeveloper's update center mechanism. For more information, see Installing Tuxedo Mainframe Transaction Publisher.

Tuxedo Mainframe Transaction Generator

Tuxedo Mainframe Transaction Generator is implemented through the JDeveloper hook. Users access this function by clicking the "Tuxedo Mainframe Transaction Generator" menu item.

By selecting this function, a graphical user interface base wizard window will be brought up to guide users to do the following things.

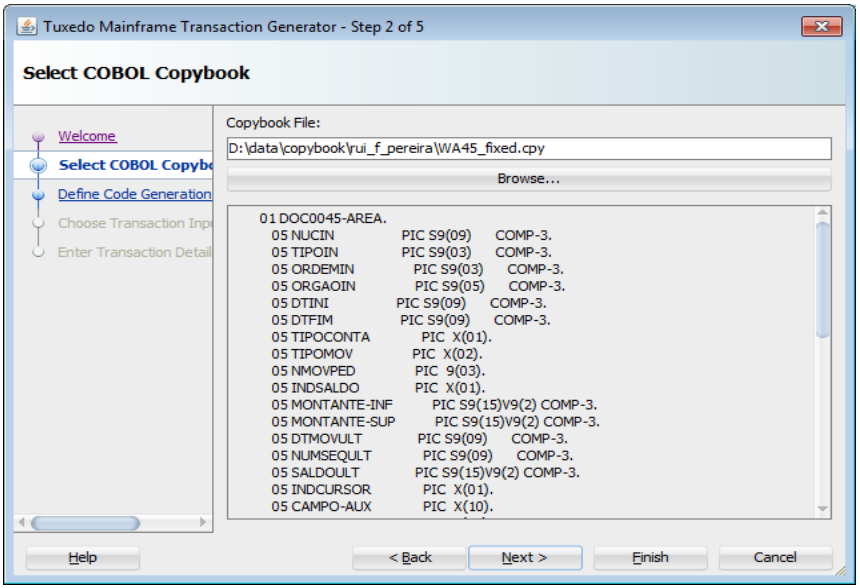
1. Select COBOL Copybook
2. Define Code Generation Details
3. Configure Transaction Input and Output
4. Enter Transaction Details

Eventually, Tuxedo Mainframe Transaction Generator generates seven artifacts that are organized in two parts.

- Generated Java code based on the COBOL copybook
- OSB related configuration data which includes WSDL, configuration for OSB Business Service, and configuration information for OSB Proxy Service

Select COBOL Copybook

The following picture shows the wizard page for selecting COBOL copybook.



Define Code Generation Details

The following screenshot shows the wizard page for defining code generation details.

The following fields are used.

Transaction ID

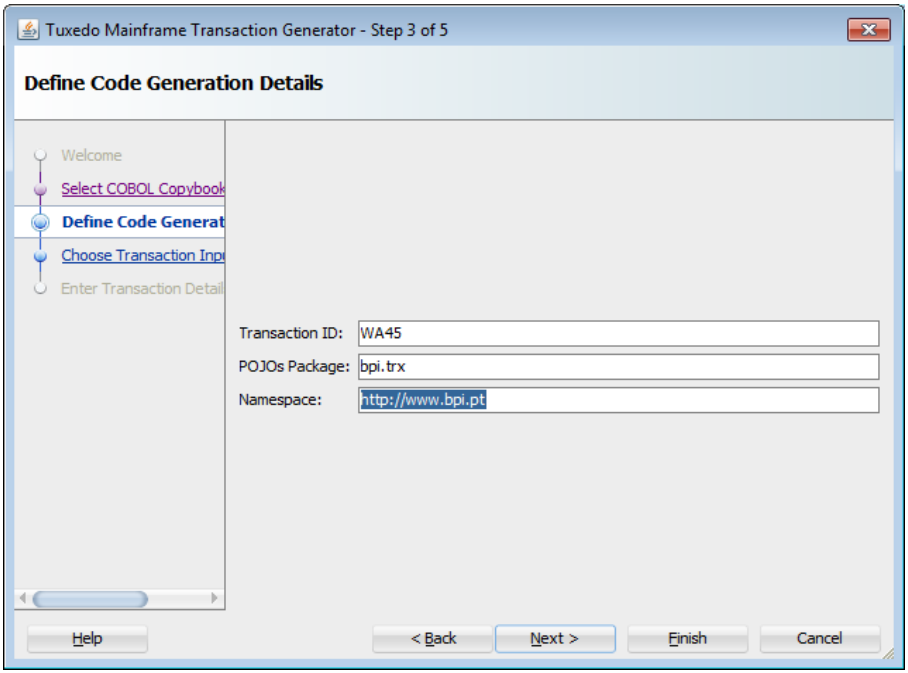
Name of the mainframe transaction. This is used in code and artifacts generation to name the OSB project, artifacts, and data mapping classes.

POJOs Package

This is used as Java package name for the mapping classes.

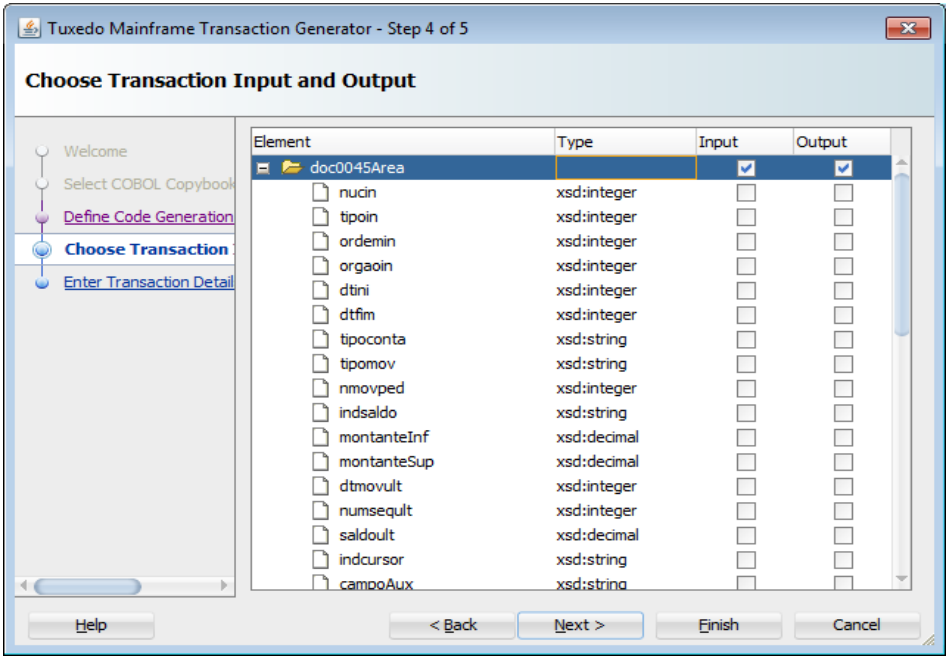
Namespace

This is used as WSDL and schema namespace in the WSDL and XSD OSB artifacts.



Configure Transaction Input and Output

The following screenshot shows the wizard page for configuring the input and output fields from the COBOL copybook.



Enter Transaction Details

The following screenshot shows wizard page for entering information needed by mainframe transaction.

The following fields are used.

Tuxedo transaction resource name
: Name of the generated Tuxedo transport/WTC import that will be generated.

Tuxedo transaction remote name
: Name of the Tuxedo service on the remote Tuxedo domain as exported from there.

Tuxedo remote domain
: ID of the remote Tuxedo/TMA domain.

Tuxedo network address
: Network address for the Tuxedo/TMA remote domain.

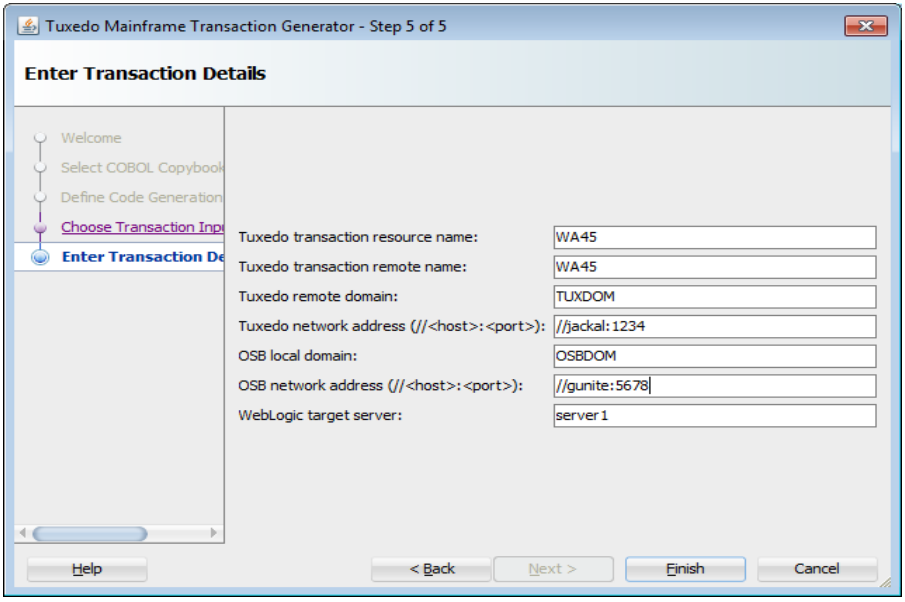
OSB local domain
: ID of the OSB domain.

OSB network address

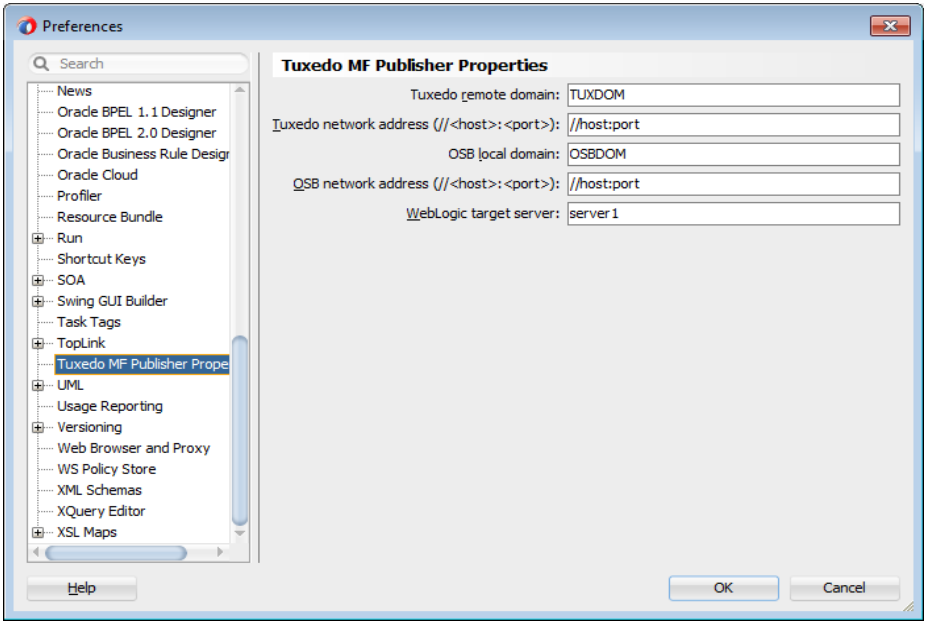
Network address of the OSB domain.

WebLogic target server

Name of the WLS server.



Users are allowed to set the defaults value for the mainframe transaction details according to user needs through the JDeveloper's "Preference" menu item from the "Tools" drop down menu.



Tuxedo Mainframe Transaction Publisher

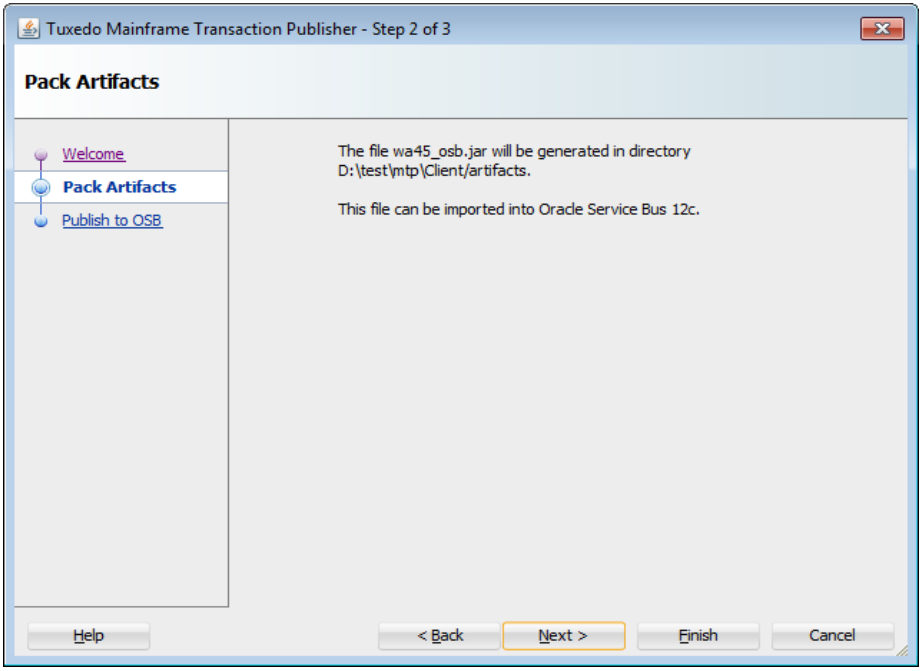
Tuxedo Mainframe Transaction Publisher is implemented through the UI hook. Users access this function by selecting the Tuxedo Mainframe Transaction Publisher menu item.

By selecting this function, a welcome wizard page will be displayed to do the following things.

1. Pack Artifacts
2. Publish to OSB

Pack Artifacts

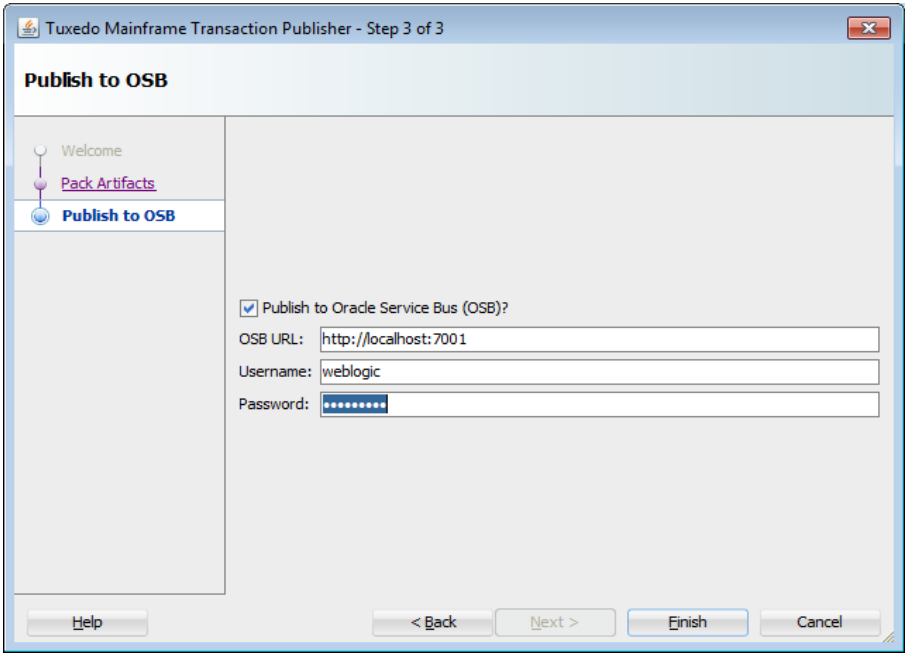
In this step, the artifacts generated by Tuxedo Mainframe Transaction Generator are packed. The following wizard page tells users the name of the packaged JAR file, and where it will be generated.



Publish to OSB

The following wizard page helps users to publish the generated artifacts to OSB. This Tuxedo Mainframe Transaction Publisher function allows users to specify the OSBs URL, administrator's name, and administrator's password.

Note: Tuxedo Mainframe Transaction Publisher allows users to manually install the OSB project by not selecting "Publish to Oracle Service Bus (OSB)?".



Installing/Uninstalling Tuxedo Mainframe Transaction Publisher

Prerequisite

To ensure successful installation of the Tuxedo Mainframe Transaction Publisher, a pristine JDeveloper should be used. Users should install the pristine JDeveloper at a new location; they should neither import any preference from other installations nor use JDeveloper to start from installer.

After installation, users use the following commands to start the JDeveloper.

- `cd $ORACLE_HOME`
- `jdeveloper/jdev/bin/jdev -clean -console`

Note: JDeveloper Studio is available for download from Oracle Technology Network.

Installing Tuxedo Mainframe Transaction Publisher

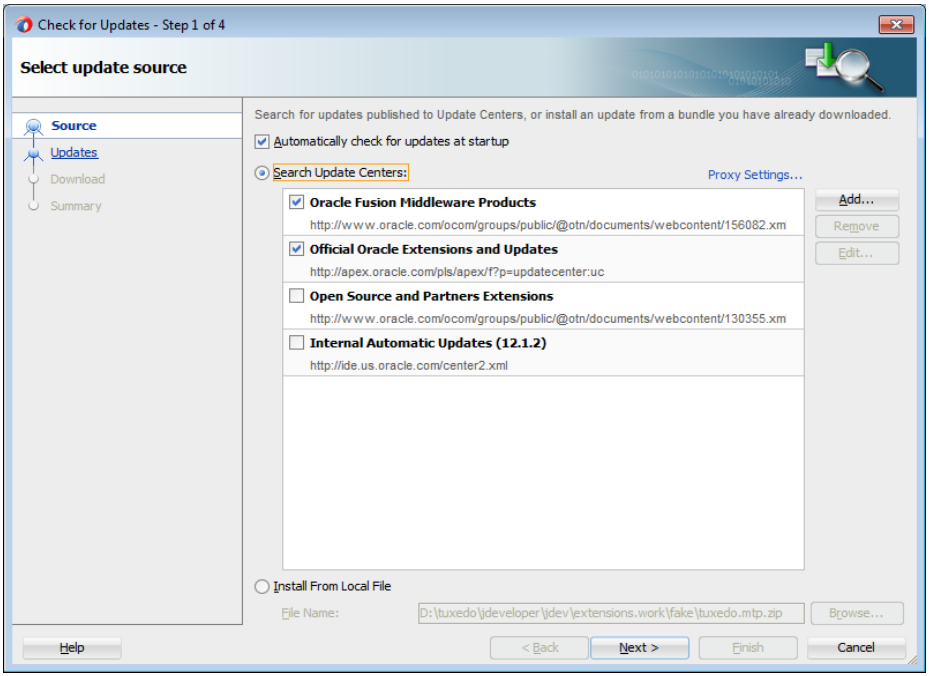
The Tuxedo Mainframe Transaction Publisher is distributed in a single zip file named "`tuxedo.mtp.update.<version>.zip`". Its current version is 12.1.2.0.

Do the following steps to complete the Tuxedo Mainframe Transaction Publisher installation.

1. Select "Install From Local File" and enter the zip file location in "File Name:" text field.
2. Click the "Next" button (and the "Summary" page shows up).
3. Click the "Finish" button to complete the installation.

After completing the installation, jar files will be installed in `MW_HOME/JDeveloper/jdev/extension/tuxedo` directory.

Note: The zip file is located in `$TUXDIR/udataobj`. To find out "`tuxedo.mtp.update.12.1.2.0.zip`", open the JDeveloper and click the "Help" menu item in the menu bar, and select "Check for Updates" from the drop down menu that is brought up.



Checking Installation Status

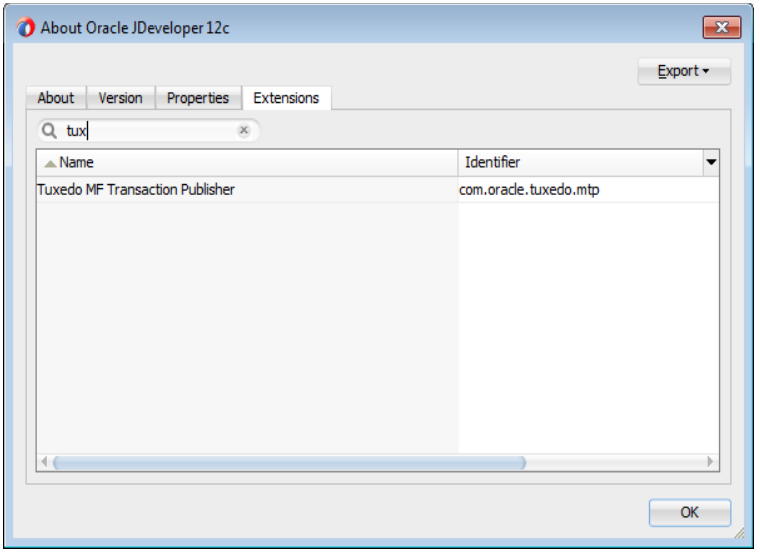
After the installation, when the updater asks to restart JDeveloper, choose not to. Then users go to the command line and re-enter `jdeveloper/jdev/bin/jdev -clean -console` to verify whether the installation is successful.

Users can check the installation status using any of the following ways.

- Using graphical user interface
- Using command lines

Using graphical user interface

Click "Help"- "About" - "Extension".



Using command lines

Listing 1 Using Command Lines to Check Installation Status

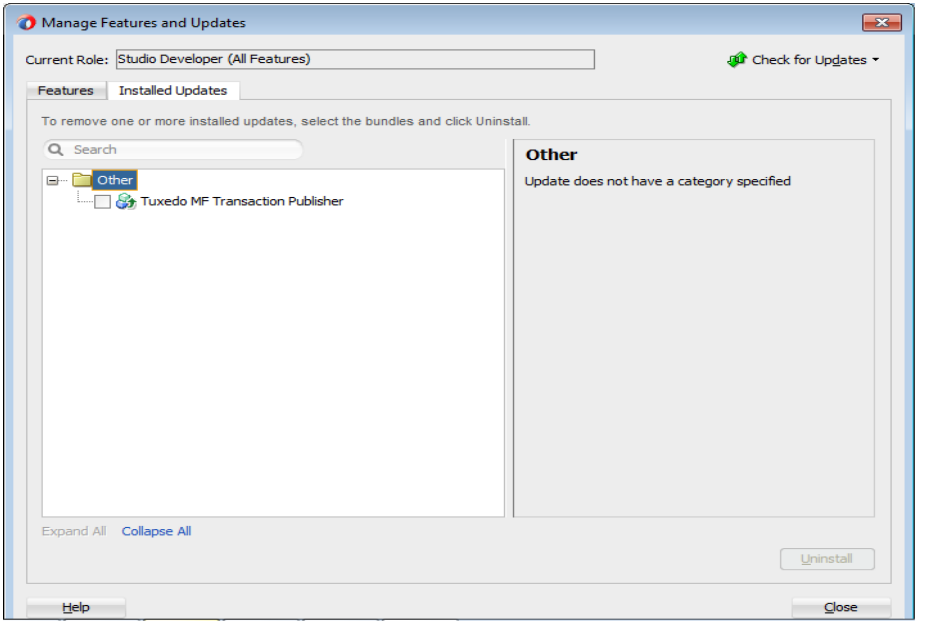
```
D:\oracle\jdeveloper\12.1.2_2>jdeveloper\jdev\bin\jdev -su -clean -console
```

```
osgi>

osgi> ss tuxedo

Framework is launched.

id      State       Bundle
927     RESOLVED    com.oracle.tuxedo.mtp_12.1.2
```

Uninstalling Tuxedo Mainframe Transaction Publisher

Do the following steps to uninstall the Tuxedo Mainframe Transaction Publisher from JDeveloper's menu bar.

1. Click the "Tools" menu item (and a drop down menu shows up).
2. Select the "Features" (and the "Manage Features and Updates" page shows up.
3. Select the "Installed Updates".
4. Select "Tuxedo MF Transaction Publisher".
5. Click "Uninstall" button to complete the uninstallation.



Installation Notes

Tuxedo Mainframe Transaction Publisher requires

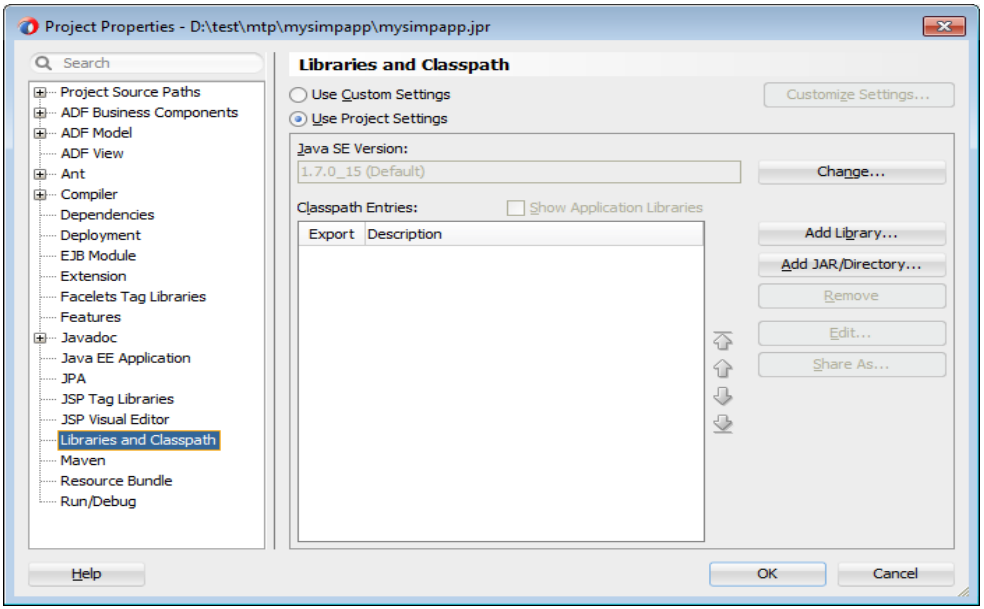
- Oracle JDeveloper 12.1.2 extension
- Oracle Service Bus (OSB) 11.1.1.7
- JDK 1.7 or above on both Oracle JDeveloper and Oracle Service Bus (OSB)

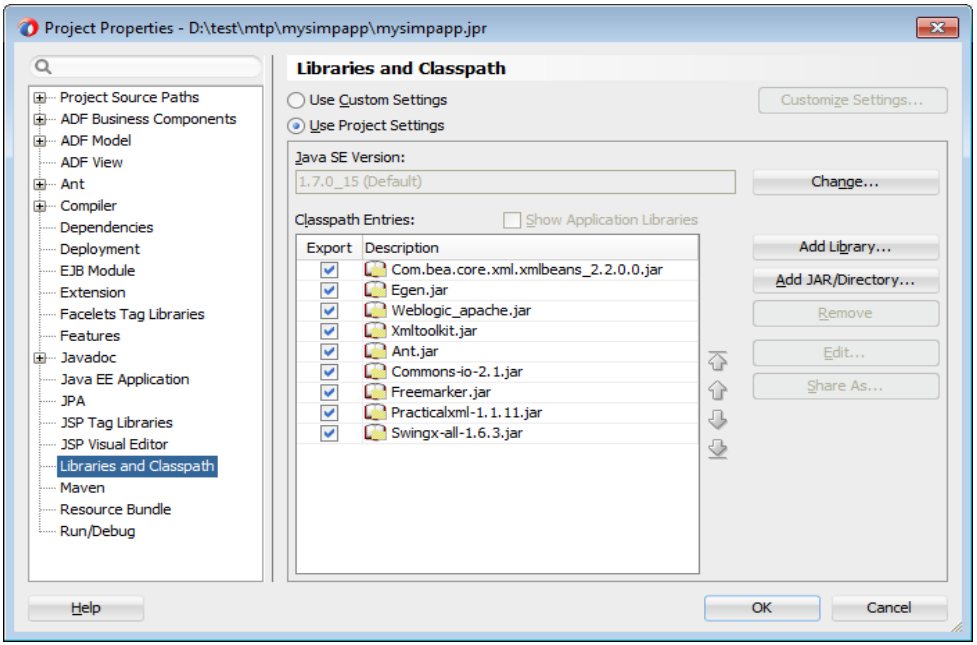
Note: When users install Tuxedo Mainframe Transaction Publisher on Oracle JDeveloper 12.1.2 extension, a matisse related exception will be reported. This exception has no impact on the use of Tuxedo Mainframe Transaction Publisher.

Setting up JDeveloper Project

Users must set up the "Library and Classpath" for every project before using Tuxedo Mainframe Transaction Publisher; otherwise, the compilation of the generated class fails.

To do the setup, right click the project to bring up context menu and select "Project Properties". Then select "Add JAR/Directory" and add the eGen libraries.


Project Properties - D:\test\mtp\mysimpapp\mysimpapp.jpr
Search
Project Source Paths
ADF Business Components
ADF Model
ADF View
Ant
Compiler
Dependencies
Deployment
EJB Module
Extension
Facelets Tag Libraries
Features
Javadoc
Java EE Application
JPA
JSP Tag Libraries
JSP Visual Editor
Libraries and Classpath
Maven
Resource Bundle
Run/Debug
Libraries and Classpath
Use Custom Settings
Use Project Settings
Customize Settings...
Java SE Version:
1.7.0_15 (Default)
Change...
Classpath Entries:
Show Application Libraries
Export
Description
Add Library...
Add JAR/Directory...
Remove
Edit...
Share As...
Help
OK
Cancel


Project Properties - D:\test\mtp\mysimpapp\mysimpapp.jpr
Project Source Paths
ADF Business Components
ADF Model
ADF View
Ant
Compiler
Dependencies
Deployment
EJB Module
Extension
Facelets Tag Libraries
Features
Javadoc
Java EE Application
JPA
JSP Tag Libraries
JSP Visual Editor
Libraries and Classpath
Maven
Resource Bundle
Run/Debug
Libraries and Classpath
Use Custom Settings
Use Project Settings
Customize Settings...
Java SE Version:
1.7.0_15 (Default)
Change...
Classpath Entries:
Show Application Libraries
Export
Description
Com.bea.core.xml.xmlbeans_2.2.0.0.jar
Egen.jar
Weblogic_apache.jar
Xmltoolkit.jar
Ant.jar
Commons-io-2.1.jar
Freemarker.jar
Practicalxml-1.1.11.jar
Swingx-all-1.6.3.jar
Add Library...
Add JAR/Directory...
Remove
Edit...
Share As...
Help
OK
Cancel

Setting up Oracle Service Bus (OSB)

Installing EGen Libraries for OSB

It is required for users to add eGen libraries to OSB's classpath by doing the following steps.

1. Create or use an existing Oracle Service Bus Domain.
2. Edit `<domain_path>/bin/setDomainEnv.sh` and eGen libraries to the classpath.
3. Restart OSB to reflect these changes in the classpath.

The eGen libraries can be extracted from the updated zip file.

Users should add the followings to `setDomainEnv.sh`.

Listing 2 Adding Information to setDomainEnv.sh

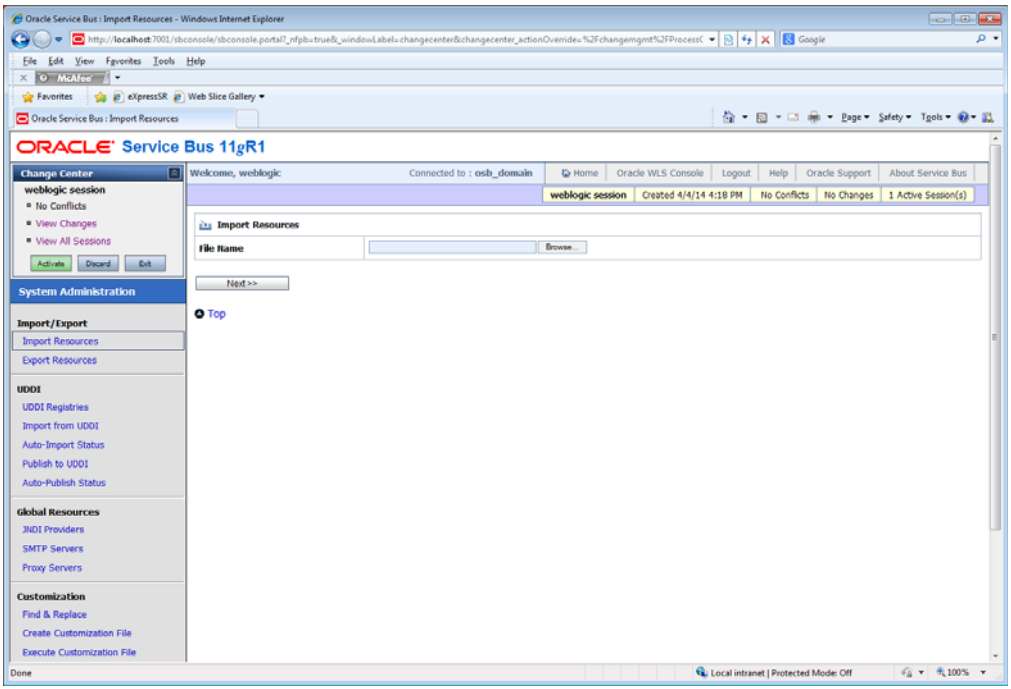
```
#
# EGen Classpath for MTP
#
BASE_EGEN_LIBS_PATH=<location of the libraries>
EGEN_CLASSPATH=${BASE_EGEN_LIBS_PATH}/com.bea.core.xml.xmlbeans_2.2.0.0.jar${CLASSPATHSEP}${BASE_EGEN_LIBS_PATH}/weblogic_apache.jar${CLASSPATHSEP}${BASE_EGEN_LIBS_PATH}/xmltoolkit.jar${CLASSPATHSEP}${BASE_EGEN_LIBS_PATH}/egen.jar
CLASSPATH="${CLASSPATH}${CLASSPATHSEP}${EGEN_CLASSPATH}"
export CLASSPATH
```

Importing Shared Resources to OSB

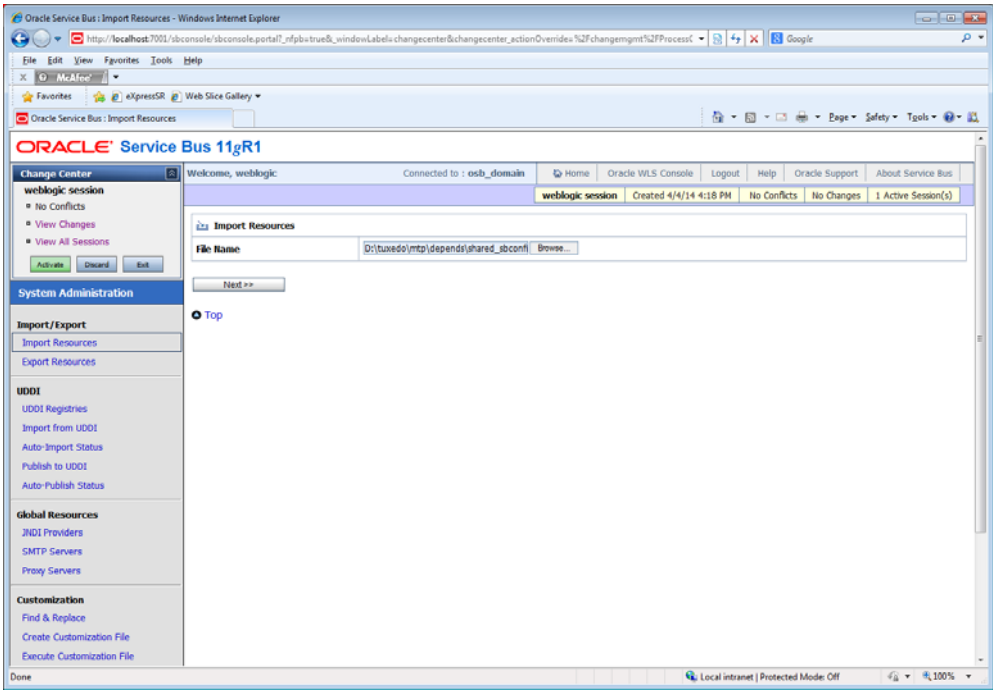
An OSB project with some shared resources is used by Tuxedo Mainframe Transaction Publisher generated OSB resources. The file with complete OSB project is in `$TUXDIR/udataobj/mtp_shared_sbconfig.jar`.

1. Use OSB's console to import this JAR.

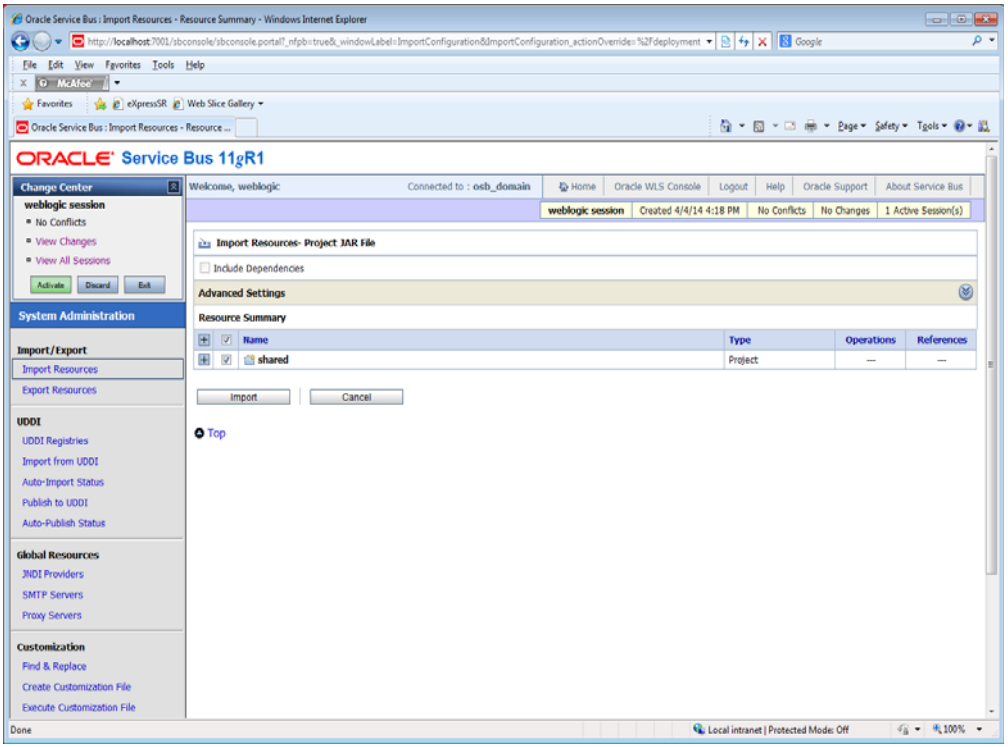
`System Administration > Import Resources`



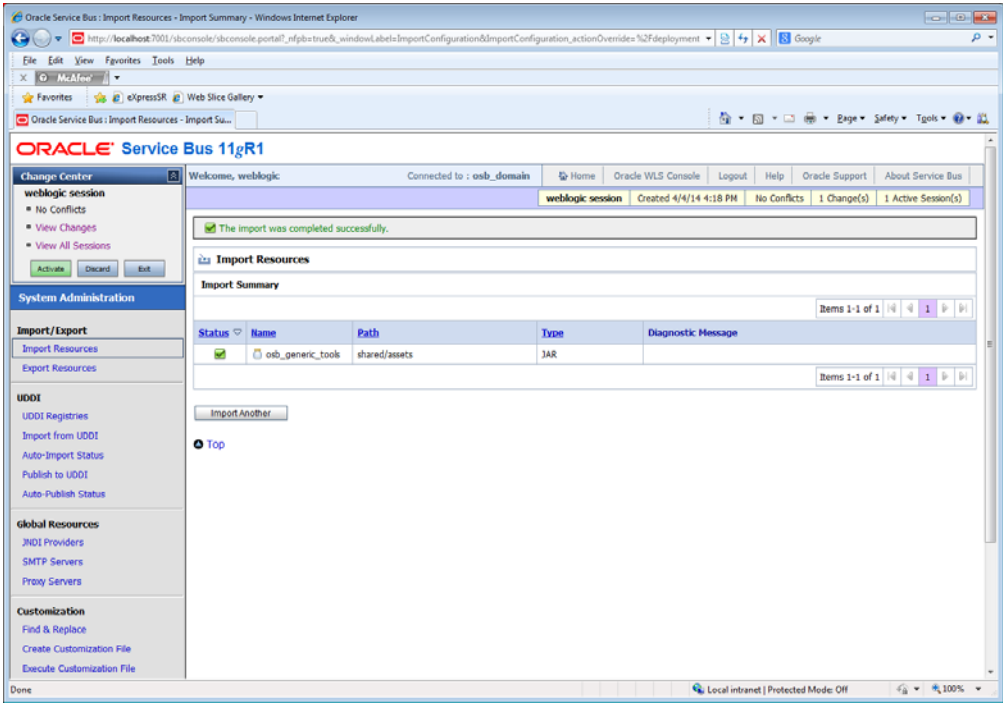
2. Enter the `mtp_shared_sbconfig.jar` location.



3. Click "Next>>" button.



4. Select "Import".



5. Click "Activate" button.
6. Click "Submit" button.
7. Check for any error or conflict and resolve them.