

Uploading Excel Spreadsheets into Oracle eBusiness Suite

Miroslav Samoilenko

October 2009

Uploading Excel Spreadsheet into Oracle eBusiness Suite

Overview

Almost any Oracle eBusiness Suite implementation faces a requirement to upload data from Excel.

Why? Sometimes, because the end users are used to Excel interface. For example, purchasing department has its own Excel template for requisitions which's been used for years and everybody likes it and wants to keep on using it. There are also situations where Excel provides a perfect environment for data manipulation and preparation. We found this to be the case almost on all our implementations of Oracle Project Accounting with monthly updates to project budgets.

Oracle's Applications Desktop Integrator (ADI) www.oracle.com/technology/tech/office/pdf/adi-userguide.pdf provides means to load information from Oracle's predefined spreadsheets for a set number of business objects, such as GL journals, GL budgets, physical inventory. When ADI meets your needs, this is the best option to go for.

Upload of complex data structures from Excel, such as sales orders or purchase requisitions, requires significant Visual Basic programming. Such Excel workbooks require strict layout structure, built in real-time validations, production database connection, and are specific to the customer. Construction cost of such a template should be measured against the productivity gains. Since an Oracle eBusiness Suite implementation team typically lacks knowledge of Visual Basic, the cost of such an extension is high. We have built several Excel templates for sales order upload and manipulation using Excel 2007 for Oracle eBusiness Suite 11.5.10.

In this article I would like to discuss an approach to load Excel spreadsheets with tabular data, such as project budgets or meter reads, into Oracle eBusiness Suite.

The simplest way to load data from Excel into an Oracle database is:

- save the Excel spreadsheet as a comma delimited file
- somehow place it on the concurrent manager tier, or anywhere where SQL*Loader is available
- run SQL*Loader to import the file into a staging table.

This approach shows the major data transformation needs, i.e. data should:

- be stored in an ASCII file
- exist on the concurrent manager tier.

It also involves significant human intervention in order to:

- Convert Excel file into CSV
- Upload to concurrent manager tier
- Run SQL*Loader

What we will discuss here is how to build a friendly user interface through which a user can upload an Excel file, and see the data loaded into a staging table using SQL*Loader. We will build this extension in such a manner that it does not depend on the nature of the incoming data and can be reused to load different Excel spreadsheets into different staging tables.

Technical Design

We build a new OAF page for the user to specify the file to upload. Since we are building a generic mechanism, the user also specifies the purpose of the upload. For example, price lists, or project budgets, or manual invoices. The list of purposes is presented as a table with name and description of the purpose for the upload. An advanced implementation can also display a URL for the file template which users can download and populate with data.

The purposes can be defined via a lookup and extracted by a query below:

```
select lookup_code
, meaning
```

Field	Type	Comments
ID	NUMBER	Primary Key, generated
PURPOSE_CODE	VARCHAR2(32)	Purpose of the upload
FILE_CONTENTS	CLOB	ASCII contents of the uploaded file

```
, description
, attribute1 template_url
from fnd_lookup_values_vl
where lookup_type = 'XXPT_FILE_UPLOAD_PURPOSE'
and enabled_flag = 'Y'
and sysdate between start_date_active and nvl(end_date_active, sysdate)
and view_application_id = 3
and security_group_id = 0
```

The data file needs to be uploaded somewhere for processing.

Since we are building an OAF page, we have two places for data processing. We can parse the file inside the page controller, and place the data into the destination tables directly. This approach allows for immediate scan of the incoming data and error reporting. The drawback is that any change to the list of files or to the file structure itself requires changes to Java code.

The only other place where we can place the data is the database. The incoming file can be placed into a customer table into a CLOB field for further processing. For that purpose, we create a table XXPT_FILE_UPLOAD_TMP with the following fields

Since the user can upload binary files such as Excel workbooks, the page controller needs to recognize those and transform into an ASCII file. In our example we will use JExcelAPI library (<http://jexcelapi.sourceforge.net/>) to transform Excel sheets into CSV stream.

We anticipate that most of uploads are for data stored in a tabular format. This assumption makes SQL*Loader our preferred data upload tool. The tool is very generic and can upload any delimited or fixed width flat file into the destination tables. Changes to the data structure can easily be addressed by changing the control file. This change can be carried by most technical consultants even without knowledge of Oracle eBusiness Suite.

However, it makes storing data in a database CLOB very impractical. We need it in a flat file somewhere where SQL*Loader can read. Since SQL*Loader is submitted as a concurrent request, we need to store the flat file in a place from which the concurrent manager can read it, for example, an output of another concurrent request.

To achieve this, we will build a Java concurrent program that is submitted immediately after the file is uploaded into XXPT_FILE_UPLOAD_TMP table. The concurrent program dumps file contents into its output stream. This concurrent program achieves our file transport goal, i.e. data file is transported to a place where it can be processed by Oracle eBusiness Suite.

The output of the concurrent program can now be passed to the SQL*Loader as the input data file. If the data file is an XML message, we can pass it to BI Publisher for processing or write our own Java concurrent program to parse and process the message. We will use the implementation of the Chain of Command described in previous articles to chain the processing concurrent program.

Building UI

First of all, we need to define business objects and explain their usage. The page we are building contains one field on top of the page where the user specifies the file to upload, and then a table below where user select the purpose for the upload.

The content for the table comes from a view object FileUploadPurposeVO which is sourced by the query from the technical design. We add to it a transient updatable attribute 'Selected' to allow for single table selection.

The content of the uploaded file is stored in the database table XXPT_FILE_UPLOAD_TMP. So, we create an entity object XXPTFileUploadTmpEO and corresponding view object XXPTFileUploadTmpVO.

We also need a primary key generator. The first thought, of course, is to define a custom sequence. I do not like this idea since the primary key here does not need to be sequential; it just needs to be unique. So, the sequence will be an extra custom database object without a very good need. We can also define or reuse a document sequence. However, this is a lot of application setups for a primary key which will live only a couple of seconds.

The approach I prefer is to use the GUID generated by the database and convert it into a number. The following query generates unique number each run you execute it.

```
select okc_p_util.raw_to_number(sys_guid()) id from dual
```

We build a view object PrimaryKeyGeneratorVO based on this query. This view object completes our data model.

There are many examples available over the internet and in the Developer's Guide on how to build an OAF page. We will skip those here. The relevant assumption is that the table which displays the upload purposes is a single selection table with an action button 'submitButton'.

This makes the page controller entry point look like this:

```
/**
 * Procedure to handle form submissions for form elements in
 * a region.
 * @param pageContext the current OA page context
 * @param webBean the web bean corresponding to the region
 */
public void processFormRequest(OAPageContext pageContext, OAWebBean webBean)
{
    super.processFormRequest(pageContext, webBean);
    OAApplicationModule am = pageContext.getApplicationModule(webBean);

    if (pageContext.getParameter("submitButton") != null)
    { processFileUpload(pageContext, webBean, am);
    }
}
```

Method processFileUpload is performing the following tasks:

- get hold of the binary file contents
- convert binary file contents into an ASCII stream
- store ASCII in the file in the database
- submit the printing request

The body of this method is:

```
protected void processFileUpload(OAPageContext pageContext, OAWebBean webBean, OAApplicationModule am)
{
    // Get hold of the binary file contents
    DataObject fileUploadData = (DataObject)pageContext.getNamedDataObject("fileUploadItem");
    if (fileUploadData == null) return;

    String fileName = (String)fileUploadData.selectValue(null, "UPLOAD_FILE_NAME");
    if (fileName == null) return;

    String contentType =(String)fileUploadData.selectValue(null, "UPLOAD_FILE_MIME_TYPE");

    BlobDomain uploadedByteStream = (BlobDomain)fileUploadData.selectValue(null, fileName);
    if (uploadedByteStream == null) return;

    // convert binary file contents into an ASCII stream
    try {
        String inputStream = streamToString(contentType, uploadedByteStream.getInputStream() );

        String purposeCode = getSelectedPurpose(am);
        if (purposeCode == null)
        {
            throw new OAException("XXRFG",
```

```

        "XXRFG_SCR0248_MISSING_PURPOSE",
        null,
        OAException.ERROR,
        null);
    }

    // Store ASCII in the file in the database
    Number primaryKey = storeStream(inputStream , purposeCode, (UploadAMImpl)am);

    int orgId = ((OADBTransactionImpl)am.getOADBTransaction()).getOrgId();
    // submit the printing request
    submitConcurrentProgram(primaryKey, purposeCode, orgId, am.getOADBTransaction().getJdbcConnection());

    } catch (IOException ex) {
        throw OAException.wrapperException(ex);
    } catch (SQLException ex) {
        throw OAException.wrapperException(ex);
    } catch (RequestSubmissionException ex) {
        throw OAException.wrapperException(ex);
    }
}
}

```

Converting Excel to CSV

There are several open source and commercial Java libraries that are capable of reading Excel workbooks. Among those are Apache POI (<http://poi.apache.org/>), JExcelAPI (<http://jexcelapi.sourceforge.net/>), JCom (<http://sourceforge.net/projects/jcom/>), ExtenXLS7 (http://www.extentech.com/estore/product_detail.jsp?product_group_id=1) to name a few. We will use JExcelAPI to recognize and process Excel files.

We assume that users are loading either Excel workbooks or ASCII files, such as XML documents or CSV files. This makes streamToString method looks like this.

```

protected String streamToString(String mimeType, InputStream inputStream) throws IOException
{
    String result;
    // check if this is an Excel spreadsheet
    if ("application/vnd.ms-excel".equalsIgnoreCase(mimeType))
    {
        try {
            result = xlsToString(inputStream);
        } catch (jxl.read.biff.BiffException ex)
        { // if not, then assume this is an ASCII stream
            inputStream.reset();
            result = streamToString(inputStream);
        }
    } else
    { // otherwise an ASCII stream
        result = streamToString(inputStream);
    }
    return result;
}

```

```
}
```

This method interprets the mimeType of the inbound file, and either reads it as an Excel workbook using method `xlsToString(InputStream)` or converts it into a string using method `streamToString(InputStream)`.

We will omit details of `streamToString(InputStream)` method, as it reads characters from the input stream and appends them to a string, and concentrate on `xlsToString(InputStream)`.

The body for this method was borrowed from here (<http://www.java-tips.org/other-api-tips/jexcel/converting-excel-documents-to-csv-files.html>). The method interprets the input stream as an Excel workbook and converts each sheet into a comma separated format.

```
private final static String CSV_SEPARATOR = ",";

private String xlsToString(InputStream stream) throws jxl.read.biff.BiffException
{
    StringWriter stringWriter = new StringWriter();
    BufferedWriter bufferedWriter = new BufferedWriter(stringWriter);
    try {
        WorkbookSettings ws = new WorkbookSettings();
        ws.setLocale(new Locale("en", "EN"));
        Workbook w = Workbook.getWorkbook(stream, ws);

        // Gets the sheets from workbook
        for (int sheet = 0; sheet < w.getNumberOfSheets(); sheet++)
        {
            Sheet s = w.getSheet(sheet);
            Cell[] row = null;

            // Gets the cells from sheet
            for (int i = 0 ; i < s.getRows() ; i++)
            {
                row = s.getRow(i);
                if (row.length > 0)
                {
                    bufferedWriter.write(formatExcelCell(row[0]));
                    for (int j = 1; j < row.length; j++)
                    {
                        bufferedWriter.write(CSV_SEPARATOR);
                        bufferedWriter.write(formatExcelCell(row[j]));
                    }
                }
                bufferedWriter.newLine();
            }
        }
        bufferedWriter.flush();
    } catch(jxl.read.biff.BiffException ex)
    {
        throw ex;
    }
    catch (Exception ex)
    {

```

```

        throw OAException.wrapperException(ex);
    }

    return stringWriter.toString();
}

```

Method `formatExcelCell` is responsible for converting the contents of a cell into the one compatible with CSV file format. Namely:

- double each double quote
- surround contents with double quotes if contents contains a double quote or comma
- convert date into DD-MMM-YYYY format

The `formatExcelCell` method body is:

```

private final static String QUOTE_STRING = "\"";
private SimpleDateFormat dateFormatter = new SimpleDateFormat("d-MMM-yyyy");

private String formatExcelCell(Cell cell)
{
    if (cell == null) return null;
    String rowCell = cell.getContents();
    // format the date
    if ( cell.getType().equals(cell.getType().DATE) )
    {
        rowCell = dateFormatter.format(((DateCell)cell).getDate());
    }
    // double each quote
    String result = rowCell.replaceAll(QUOTE_STRING, QUOTE_STRING+QUOTE_STRING);
    // surround with quotes if comma or quote is present
    if (result.indexOf(CSV_SEPARATOR) >0 | | result.indexOf(QUOTE_STRING) >0)
    {
        result = QUOTE_STRING + result + QUOTE_STRING;
    }
    return result;
}

```

Here we reached the point when the incoming stream is tested to be an Excel workbook, and if so, converted into a comma separate stream. Otherwise, it is assumed that the incoming file is an ASCII stream.

Storing CLOB in Database Table

Once we have the input stream converted into an ASCII string, we need to store it in the custom table. To perform this operation, we need to generate primary key, and extract the purpose for the upload.

We already discussed the query which generated globally unique primary key, and creation of a view object `PrimaryKeyGeneratorVO` based on this query. The method which generates the primary key becomes:

```
protected Number generatePrimaryKey()
{
    OAViewObject viewObject = getPrimaryKeyGeneratorVO();
    viewObject.setMaxFetchSize(1);
    viewObject.executeQuery();
    return (Number) (viewObject.first().getAttribute("Id"));
}
```

In order to determine the select purpose, we need to walk through each row of purposes in the current range, and locate the one with checked attribute 'Selected', since this was the transient attribute designated to work with table single selection. The method is:

```
public String getSelectedPurpose()
{
    OAViewObject viewObject = getFileUploadPurposeVO();
    Row[] rows = viewObject.getAllRowsInRange();
    if (rows.length > 0)
    {
        for(int i=0; i<rows.length; i++)
        {
            Row row = rows[i];
            if ("Y".equals(row.getAttribute("Selected"))) {
                return (String)row.getAttribute("LookupCode");
            }
        }
    }
    return null;
}
```

Now, we are ready to store data in the database. We will need the purpose of the upload later when we submit the concurrent program to print the file contents in its output. So, we extract the purpose separately and pass as a parameter to the storage procedure, which now looks like:

```
public Number storeStream(String inputStream, String purposeCode)
{
    OAViewObject viewObject = am.getXXPTFileUploadTmpVO();
    viewObject.setMaxFetchSize(0);

    ClobDomain myClob = new ClobDomain();
    myClob.setChars(inputStream.toCharArray());
    OARow row = (OARow)viewObject.createRow();
    row.setAttribute("FileContents",myClob);
    Number primaryKey = generatePrimaryKey();
    row.setAttribute("PurposeCode", purposeCode);
    row.setAttribute("Id", primaryKey);
    viewObject.insertRow(row);
    am.getTransaction().commit();
    return primaryKey;
}
```

The method returns the primary key of the newly created record. We will later pass this key to the concurrent program that prints the output of the CLOB into its output.

Writing Java Concurrent Program

Any Java class that implements interface `oracle.apps.fnd.cp.request.JavaConcurrentProgram` can be registered as a concurrent program.

The only method which needs to be implemented is:

`public void runProgram(oracle.apps.fnd.cp.request.CpContext cpcontext)`
CpContext provides developer with all attributes of a PL/SQL concurrent program such as output file, log file, and return status plus a database connection. A typical implementation of this method looks like

```
public void runProgram(CpContext cpcontext) {
    try {
        // read parameters
        // execute business logic
        cpcontext.getReqCompletion().setCompletion(ReqCompletion.NORMAL, "Request Completed Normal");
    } catch (Exception ex) {
        // report exception
        cpcontext.getReqCompletion().setCompletion(ReqCompletion.ERROR, "Error building output file");
    } finally {
        cpcontext.releaseJDBCConnection();
    }
}
```

Though this template is small, it does illustrate the key differences between Java and PL/SQL concurrent program.

By default, Java concurrent program completes with an error, unlike PL/SQL program which completes successfully. Hence, it is imperative to set up not only completion in error, but also when program completes successfully.

At the end of the execution of the Java concurrent program, you must release JDBC connection back to the pool.

Reading Parameters

Concurrent program parameters are accessible via method `cpcontext.getParameterList`. This method returns an instance of `oracle.apps.fnd.util.ParameterList` class. This class provides an Enumeration interface to access all the parameters passed to the program in the order defined during the concurrent program registration. This class has one interesting drawback. You can read the parameters only once.

Each concurrent program parameter is represented by an instance of class `oracle.apps.fnd.util.NameValueType`. It is clear from the name of the class that it provides name, type and string value of the parameter. It means that if you have a date value as the parameter to the concurrent program, it will be available to the Java concurrent program as a canonical string.

I found it useful to have a utility that converts the `ParameterList` into a `Map`.

```
static public Map convertParameters(ParameterList parameterList)
{ Map result = new HashMap();
  while( parameterList.hasMoreElements() ) {
    NameValueType nameValueType = parameterList.nextParameter();

    if (nameValueType.getValue() != null)
    {
      result.put(nameValueType.getName(), nameValueType.getValue());
    }
  }
  return result;
}
```

Map provides a better controlled access to the list of parameters. The method can be enhanced to recognize the type of the value and convert them from String into BigDecimal or Date.

Reporting Exception

Exception.printStackTrace() does not work here!!!

If you really want the user to see the exception stack, you need to print it into the oracle.apps.fnd.cp.request.LogFile. An instance of this class is available via cpcontext.getLogFile(). This class provides an OutputStream like interface, though it is not an implementation of java.io.OutputStream. LogFile provides a set of methods to write strings into the output stream
write(String message, int level)

Here, level specifies the debugging level which can be set from LogFile.STATEMENT to LogFile.EXCEPTION.

In order to report exception to the log file, you can use the following code

```
StringWriter writer = new StringWriter();
ex.printStackTrace(new PrintWriter(writer));
logFile.writeln(writer.toString(), logFile.EXCEPTION);
```

Writing Output

The key task of the concurrent program which we are writing is to read the CLOB that contains the file contents and write it into its output stream. The output of a concurrent program can later be accessed by SQL*Loader, other concurrent program or, even, OAF web pages and services outside of Oracle eBusiness Suite.

To access the database, we need to get hold of the database connection. An instance of the connection can be received from cpcontext.getJDBCCConnection().

The following method retrieves the CLOB from the database as an InputStream which is then converted into a String:

```
protected String getOutput(Connection connection, BigDecimal primaryKey) throws SQLException, IOException
{
  String statement = "select file_contents from xxpt_file_upload_tmp where id = :1";
  String result;
  PreparedStatement stmt = null;
```

```

ResultSet resultSet = null;
try {
    stmt = connection.prepareStatement(statement);
    stmt.setBigDecimal(1, primaryKey);
    resultSet = stmt.executeQuery();
    resultSet.next();
    result =streamToString( resultSet.getAsciiStream(1) );
    resultSet.close();
    resultSet = null;
    stmt.close();
    stmt = null;
} finally
{
    if (resultSet != null)
    {
        try { resultSet.close(); } catch (SQLException ex) {}
    }
    if (stmt != null)
    {
        try { stmt.close(); } catch (SQLException ex) {}
    }
    try { connection.commit(); } catch (SQLException ex) {}
}
return result;
}

```

In this code streamToString method read the InputStream into a String.

An instance of oracle.apps.fnd.cp.request.OutFile get be obtained using methof cpcontext.getOutFile(). This class provides OutputStream like interface, though it is not an implementation of java.io.OutputStream. Writing a string into the output is as simple as:

```
cpcontext.getOutFile().write(fileContents);
```

Submit Concurrent Program from Java

Oracle provides Java classes to submit a concurrent program. Definitely, one can call PL/SQL procedure directly using JDBC connection, though the Java wrapper is much nicer and easier to use. Class oracle.apps.fnd.cp.request.ConcurrentRequest represents submission of a concurrent request.

You can setup optional layouts, notifications, schedule and other attributes available through FND_REQUEST package.

Constructor of ConcurrentRequest class accepts database connection instance as a parameter. Signature of the method that submits the concurrent request for execution
submitRequest(String application, String program, String description, String startTime, String subrequest, Vector parameters)

mimics parameters of the corresponding PL/SQL procedure FND_REQUEST.SUBMIT_REQUEST. Here, parameters is the vector of String in the order the parameters are specified during the concurrent program registration. If one of your parameters is a Date, it is your responsibility to convert it into the canonical string.

So, the code that submits the concurrent program may look like:

```
protected void submitConcurrentProgram(Number primaryKey, String purposeCode, int orgId,
Connection connection) throws IOException, SQLException, RequestSubmissionException
{
    ConcurrentRequest request = new ConcurrentRequest(connection);
    Vector param = new Vector();
    param.add(primaryKey.stringValue());
    param.add(purposeCode);
    param.add(String.valueOf(orgId));

    int reqId = request.submitRequest("XXPT", "XXPTCLOBOUTPUT", "Print CLOB", null, false, param);
    connection.commit();

    MessageToken[] tokens = { new MessageToken("REQUEST", String.valueOf(reqId)) };
    OAException confirmMessage = new OAException("XXPT",
        "XXPT_REQUEST_SUBMIT_CONF",
        tokens,
        OAException.CONFIRMATION,
        null);
    throw confirmMessage;
}
```

As you can see, upon successful submission of the concurrent program, we raise a confirmation exception with the request ID.

Summary

In this article we covered the following topics:

- Technical design for file upload extension
- Database design
- Construction of OAF user interface
- Reading Excel spreadsheet using JExcelAPI
- Storing CLOB in a database
- Writing and submission of Java concurrent program

The last step that is deliberately not covered in this article is using the Chain of Command design pattern to chain the Java concurrent program with a SQL*Loader process to upload the comma delimited file into the database for further processing.

Please, read about this design pattern in our previous posts and articles.

The Chain of Command implementation described gives you a powerful tool to declaratively define post processor to concurrent programs. You can choose to hard code the calls inside your existing code, and submit SQL*Loader concurrent program right from the Java concurrent program that prints out the uploaded file.

This is our goal to show the ways and let you choose the one fit for you.