Page 1 October 2006 Volume II Issue 4
ORAtips Journal

Let UNIX Drive Your Oracle® Interfaces

By Kevin Ellis

***Editor's Note:** Kevin Ellis continues to explore the use of Oracle's Interface Tables. His current article (also an Oracle Open World presentation) discusses data processing and concurrent manager management using FTP sessions and UNIX shell programs with good old fashion e-mail to load journal entries into Oracle General Ledger.*

Background

Interfaces comprise the bulk of most information systems. The chances are that at your place of employment, not every piece of information is centralized. In fact, it may be spread over a series of unrelated systems. How do these systems communicate with each other? That is where the interface comes into play. According to the Merriam-Webster dictionary, an interface is a place at which independent and often-unrelated systems meet and act on or communicate with each other. Oracle Applications is no different.

Interfaces come in many flavors. An interface could involve reconciling items for journal entries, check balancing, or handling approval hierarchies from a foreign HR system. For this article, I will discuss a simple interface that loads journal entries into Oracle General Ledger.

Originally this interface was deployed via a request set. The first stage executed a UNIX shell program that would initiate an FTP session to pull the file from Legacy to UNIX. The next stage would execute a SQL*Loader session to load the data into a staging table. The third stage would execute a SQL*Plus session to execute a PL/SQL program that would process the data in the staging table and move the results to Oracle's GL Open Interface Table (gl_interface). The fourth stage would execute a couple of concurrent programs that were built in-house to automate the Journal Import process. The final stage initiated another job that would send e-mail to the caller, notifying him or her when the request set was done.

But wouldn't it be nice to find the problem immediately?

Anyone who has experience working with request sets (especially from the support side) can attest to the fact that the job will generate multiple request IDs within the concurrent manager. Users would call me about a problem with the interface. In response, I would ask them for the corresponding request ID. Sometimes I would get only the concurrent program that completed in error. Sometimes it would be the Journal Import job that had completed in error (usually from a period not being opened). In many cases, the automated Journal Import job did not give us any clue as to the real problem with the interface. Even the concurrent programs that execute have parent processes, and the parent processes point back to the reset set. You can quickly have a basketful of request IDs to sort through. Most of the problems are simple to solve, once you find them. But wouldn't it be nice to find the problem immediately? Wouldn't it be nice to see one log generated from a single concurrent request that logged the entire interface process?

I sometimes work with contractors, and there's one whom I remember very well. She was given the task of putting together an interface, and I was expecting to see a request set as the deliverable. The deliverable was to cut an extract out of Oracle AP and send it to another remote system. This meant that a UNIX shell program would be needed to send the file to the remote system via an FTP session. Likewise, a PL/SQL program would be required to pull data from the database and create the file to be sent. To me, this would mean two separate concurrent programs executed in sequence via a request set. To my surprise, the deliverable was a single concurrent program that executed a single UNIX Shell Program. It worked, and it worked fast.

Given to curiosity, I investigated what had been deployed. As it turned out, the PL/SQL program was called from within the UNIX Shell program prior to the FTP session being called. Simply put, the contractor used a

ORAtips © 2006 Klee Associates, Inc.

Page 2 ORAtips Journal October 2006 Volume II Issue 4

UNIX Shell program as the driver for the interface. Given these facts, I started to experiment with other possibilities. Could SQL*Loader be called from within a UNIX Shell Program? How about send E-mail? Can parameters be passed to a UNIX Shell Program and how do you retrieve them for processing? These and other questions were researched. As it turned out, all of these things could be done. The power of this approach was twofold: Reduction of request IDs, and reduction of processing time.

This article will demonstrate how you can put together an interface using a single concurrent program. It will demonstrate this approach from a case study involving a SWAT (Strengths, Weaknesses, Achievements, and Threats) analysis, followed by how to implement the approach.

Options

There are two primary options for deploying an interface in Oracle Applications: a concurrent program or a request set. A request set is a collection of stages. The same concurrent program could be executed in each of these stages. Why might you do this? Perhaps this option would be best if there are different parameters to be used for each stage. Also, multiple concurrent programs can be executed within a single stage. There are all sorts of possibilities with deploying an interface via a request set.

From the developer's viewpoint (or at least mine), I prefer the concurrent program. I execute a single concurrent program and if there is a problem, all I have to worry about is one (in most cases) request ID. Using the request ID, I can view the associated log, which documents every stage of the interface process. I don't have to go searching for various logs in order to put the entire picture together.

There are two primary options for deploying an interface in Oracle Applications: a concurrent program or a request set.

I decided to prove that the concept was really viable by redeploying an existing interface (available as a request set) using the new concept (single concurrent program) and measuring the differences. From here, I will discuss the results of my analysis. Construction of the concept will be discussed under "Implementation".

Every option has its strengths and weaknesses. As for concurrent programs and request set, this is what I have uncovered (see Figure 1):

A lot of these conclusions are based on tests that I have performed. Concerning the tests, I ran two series: one for a small batch of data and one that was significantly larger. Given, data volume can alter from one run to another and data is different from one company to another. In any event, the demonstration is made to show a simple difference between running an interface using a single concurrent program verses a request set. Figure 2 provides time results using a small batch of data.

Concurrent Program	
Strengths	**Weaknesses**
Interface stages are programmed	Moderate knowledge of UNIX shell scripting
Quicker response in resolving Issues	Control bypasses the concurrent manager
Shorter execution duration	
Fewer concurrent requests generated	
Fewer front-end setups	
Excluding UNIX driver program, scripts can have various locations	
Better error trapping via messaging	
Request Set	
Strengths	**Weaknesses**
Stages can be deployed via the application	Takes more time to run
Users have control of the request set stages	More setups on the application side
Control retained within the concurrent manager	Documentation of application setup is complicated
	Difficulty in identifying background-spawned concurrent request

Figure 1: SWAT

ORAtips © 2006 Klee Associates, Inc.

	Request Set	Concurrent Program	Difference
Test #1	100	68	32
Test #2	82	51	31
Test #3	59	49	10
Test #4	44	27	17
Test #5	72	48	24
Test #6	129	27	102
Test #7	72	46	26
Test #8	91	38	53
Test #9	68	41	27
Test #10	87	40	47
Total Time	804	435	369
Avg Time	80.4	43.5	36.9
Factor	1.85	0.54	8.48
Processing Time	84.83%	45.90%	
	longer	shorter	
Requests per run	16	2	14

Figure 2: Test #1

The sample consisted of 1014 records.

In this test scenario, I used a batch of 1014 records. I ran the test for each option 10 times. What I discovered was that, on average, the concurrent program was shorter in duration by 46% compared with the request set. In other words, if a request set takes 100 units of time to complete, the concurrent program approach could accomplish the same task in only 54 units. This is a significant amount of processing time that is free to be used on other tasks. Suppose you had multiple batches to process; you could almost complete two batches using the concurrent program method in the time it takes to run one batch using the request set method.

To make sure that the results were accurate, I decided to run the tests again. Except this time, the data batch would be larger. In this scenario, the batch would increase to 40236 records. Figure 3 displays the results of this new test.

The sample consisted of 40,236 records

In this test, the gap between completion rates has narrowed. Still, the concurrent program completes the processing in roughly 87.2% of the time it would take the request set to complete.

After reviewing the strategy closer, I determined that data (regardless of whether using a single concurrent program or a request set) takes a set amount of time to process for each stage of the interface cycle. The major difference is that with a request set, each stage kicks off a separate concurrent request. These individual concurrent requests have to wait for the concurrent manager to execute them. The wait time between

	Request Set	Concurrent Program	Difference
Test #1	222	193	29
Test #2	247	197	50
Test #3	244	218	26
Test #4	265	214	51
Test #5	246	213	33
Test #6	209	184	25
Test #7	223	206	17
Test #8	242	210	32
Test #9	239	214	25
Test #10	236	220	16
Total Time	2373	2069	304
Avg Time	237.3	206.9	30.4
Factor	1.15	0.87	1.47
Processing Time	14.69%	12.81%	
	longer	shorter	
Requests per run	16	2	14

Figure 3: Test #2

ORAtips © 2006 Klee Associates, Inc.

ORAtips

On Development

concurrent requests can be changed. But even if you make the wait time less, there will be a pause between executing the various stages within a request set. As the data volume increases for an interface, the difference between using the quicker concurrent program option to the request set will narrow. The concurrent program option will always be faster since there is no wait time (unless programmed in using the sleep command) between executing stages. That leaves us with the decision whether to go with the request set option, allowing you to set up multiple stages via Oracle Applications, or use a single concurrent program, requiring UNIX scripting knowledge to execute all of the stages? If your users are not involved or don't care about the various stages in which a process is executed, you might want to explore the possibility of deploying a single concurrent program that executes a UNIX scripting script. Regardless of the data volume, the concurrent program will always finish before the request set.

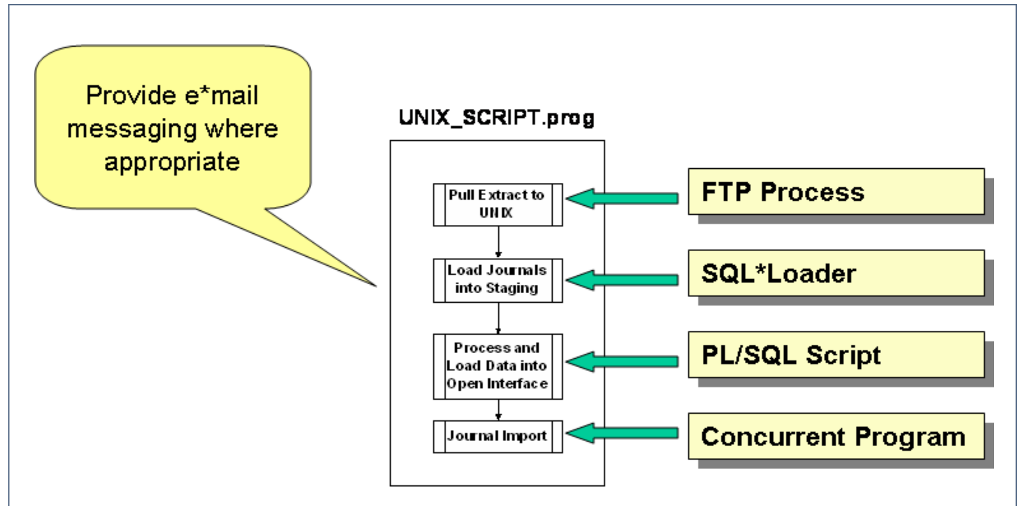


Figure 4: Design

But there is another way of executing from the UNIX shell using a utility provided by Oracle called CONCSUB

Also, you only have to be responsible for one primary request ID, whereas the request set will generate multiple request IDs.

Implementation

Let's consider that you want to explore the single concurrent program approach. Let me show you how to go about developing an interface (start to finish) driven by a UNIX shell program called from a single concurrent program.

Design

First, let's put together a design of an interface. That design is listed in Figure 4. OK, I told you that we would only be working with one request ID. Well, that is not totally true, as in this case we will be working with two. Still, this is far fewer IDs than what would be produced via a request set. One of the request IDs will be for executing the single concurrent program. The other will be for executing an Oracle-supplied job called "Journal Import". For those of you not familiar with Journal Import, it is the utility used in General Ledger that imports journal entries from remote data sources. Journal Import is usually executed from within Oracle Applications. But there is another way of executing from the UNIX shell using a utility provided by Oracle called CONCSUB. You can find more information about this utility on MetaLink. Later, I will provide you with the details for executing Journal Import via CONCSUB. Executing Journal Import from the background will generate the other request ID.

The Driver Program

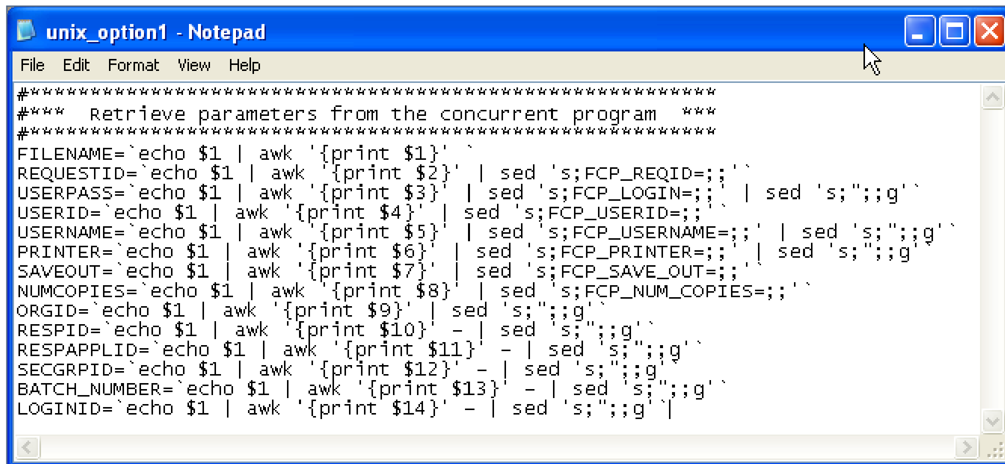
Let's talk a little about the design of this concept. Using a UNIX shell program, we can execute FTPs, Oracle's SQL*Loader, Oracle's SQL*Plus, and Oracle CONCSUB. With all of this, we have complete control over pulling the data, loading it into a staging table, processing it, and importing it into the production tables used by Oracle Applications. In our case, we will be importing journal entries from a remote system and loading them into General Ledger.

There are two approaches to setting up the UNIX shell program. The two approaches come down to how you process parameters sent from Oracle Applications to the UNIX shell program.

The first approach is by retrieving all of the parameters using AWK. AWK is basically used for parsing

ORAtips © 2006 Klee Associates, Inc.

through a string of data. In this case, all of the data sent from Oracle Applications is received via the first UNIX parameter. You have to use AWK to parse through this string in order to get all of the necessary parameters used by the interface. The code is very hard to read and time consuming to program. But, it gets the job done. Figure 5 demonstrates this approach.



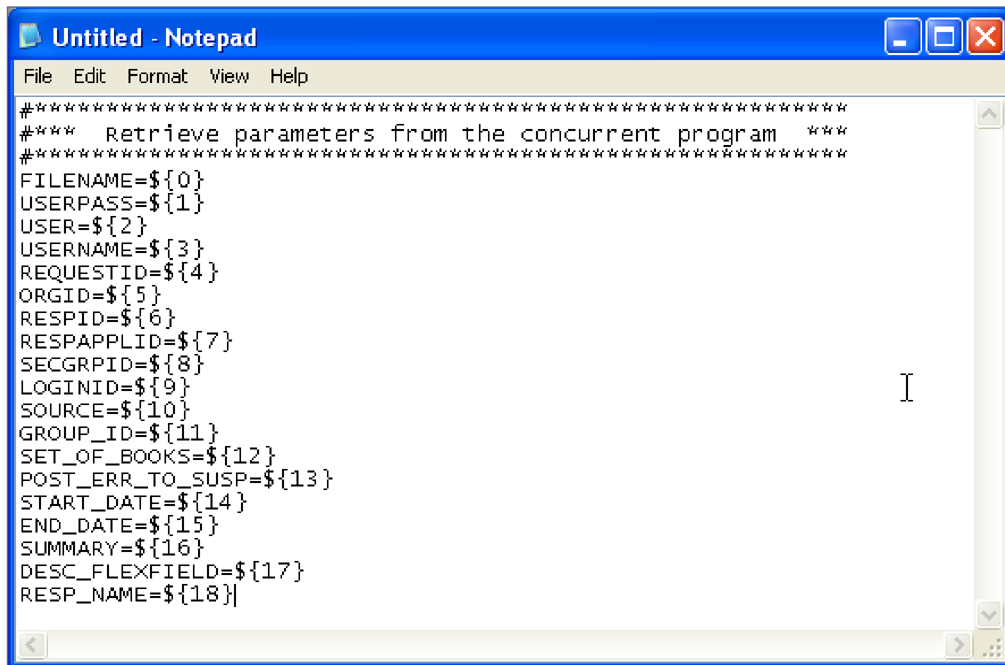
```
#*************************************************************
#***  Retrieve parameters from the concurrent program  ***
#*************************************************************
FILENAME=`echo $1 | awk '{print $1}' `
REQUESTID=`echo $1 | awk '{print $2}' | sed 's;FCP_REQID=;;'`
USERPASS=`echo $1 | awk '{print $3}' | sed 's;FCP_LOGIN=;;' | sed 's;";;g'`
USERID=`echo $1 | awk '{print $4}' | sed 's;FCP_USERID=;;'`
USERNAME=`echo $1 | awk '{print $5}' | sed 's;FCP_USERNAME=;;' | sed 's;";;g'`
PRINTER=`echo $1 | awk '{print $6}' | sed 's;FCP_PRINTER=;;' | sed 's;";;g'`
SAVEOUT=`echo $1 | awk '{print $7}' | sed 's;FCP_SAVE_OUT=;;'`
NUMCOPIES=`echo $1 | awk '{print $8}' | sed 's;FCP_NUM_COPIES=;;'`
ORGID=`echo $1 | awk '{print $9}' | sed 's;";;g'`
RESPID=`echo $1 | awk '{print $10}' - | sed 's;";;g'`
RESPAPPLID=`echo $1 | awk '{print $11}' - | sed 's;";;g'`
SECGRPID=`echo $1 | awk '{print $12}' - | sed 's;";;g'`
BATCH_NUMBER=`echo $1 | awk '{print $13}' - | sed 's;";;g'`
LOGINID=`echo $1 | awk '{print $14}' - | sed 's;";;g'`
```

Figure 5: Parameters via AWK

Here, all of the parameters passed from Oracle Applications can be retrieved from the UNIX shell program by accessing UNIX parameter $1. The key is in knowing the order in which the parameters are listed. By default, Oracle passes parameters that are not visible when setting up a concurrent program: Filename, Request ID, User Password, User ID, User Name, the Printer for the Concurrent Program, the Save Out option, and the Number of Copies to Print. In order to obtain the values without all the other garbage associated with it, I use "sed", which allows me to replace characters. In this case, unwanted characters are basically eliminated or replaced with null. Starting at the 9th position in $1 are the parameters I actually created in association with the concurrent program. In this case (and this is from another, unrelated interface), there were concurrent program parameters for Org ID, Responsibility ID, Responsibility Application ID, Security Group ID, Batch Number, and Login ID. Basically, values 9 through 14 were the six parameters I set up for the concurrent program in question. It should come without question that this is rather hard to read and understand unless you are familiar with AWK and sed.

The next option (used in this case study and the one that I recommend) involves creating a UNIX Link. To use the UNIX Link, the UNIX shell program must have a ".prog" extension. This may seem like a lot of extra work, but it really isn't; especially when you consider how much easier it will be to retrieve parameters from the concurrent program, plus future maintenance.

First, let's investigate the UNIX shell program, listed in Figure 6.



```
#*************************************************************
#***  Retrieve parameters from the concurrent program  ***
#*************************************************************
FILENAME=${0}
USERPASS=${1}
USER=${2}
USERNAME=${3}
REQUESTID=${4}
ORGID=${5}
RESPID=${6}
RESPAPPLID=${7}
SECGRPID=${8}
LOGINID=${9}
SOURCE=${10}
GROUP_ID=${11}
SET_OF_BOOKS=${12}
POST_ERR_TO_SUSP=${13}
START_DATE=${14}
END_DATE=${15}
SUMMARY=${16}
DESC_FLEXFIELD=${17}
RESP_NAME=${18}
```

Figure 6: Parameters via UNIX Link

ORAtips © 2006 Klee Associates, Inc.

Page 6 October 2006 Volume II Issue 4 ORAtipsJournal

As you can see, it is much easier to identify the parameters coming in without having to sort through all of the AWK code. However, I must mention that the first six parameters (FILENAME, USERPASS, USER, USERNAME, REQUESTID, and ORGID) are passed to the UNIX shell program regardless. Hence, the user-defined parameters in the concurrent program can be referenced starting with the 6th parameter passed to the UNIX shell program. For instance, Organization ID can be referenced via ${5}. Likewise, Responsibility ID can be referenced via ${6}, and so on. This is a much easier way to deploy and maintain. However, the side effect is that a UNIX Link must be established.

Creating the UNIX Link is not very difficult. It can be accomplished by executing the code listed in Figure 7.

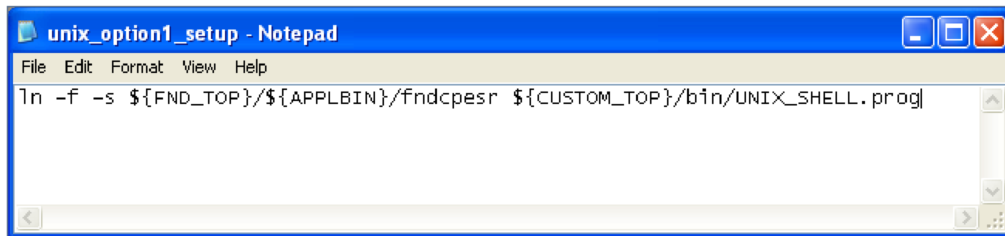
The "–s" option tells UNIX to create a symbolic link. This method actually links the UNIX shell program (UNIX_SHELL.prog located under ${CUSTOM_TOP}/bin) to FNDCPESR; an Oracle-provided program that Oracle Applications uses to run UNIX shell programs for easier parameter processing. The "–f" causes the ln command to replace any destination paths that already exist. Using this method eliminates the need to use a separate scripting language (like AWK) to parse the parameters out of the ${0} variable. One important note: if you clone your environments regularly, the links will also need to be recycled; otherwise, they point to the cloning source, causing the concurrent program to complete in error. This error is sometimes hard to identify. But after seeing it happen several times, it is usually one of the first things I look for before proceeding further. Additionally, make sure you have "execute" access on the ".prog" version of your UNIX shell program for both the owner and group.

Command Prompt to UNIX Shell Program

Now that I have demonstrated how to set up the UNIX shell program, it is time to discuss the implementation features. If you can execute something from the command prompt, more than likely it can be scripted and executed via a UNIX shell program. So, let's dive into a few concepts that you can use to accomplish the tasks associated with Figure 4. One note, I will be referencing several parameters, all of which originate from the UNIX shell program listed in either Figure 5 or Figure 6.

FTP

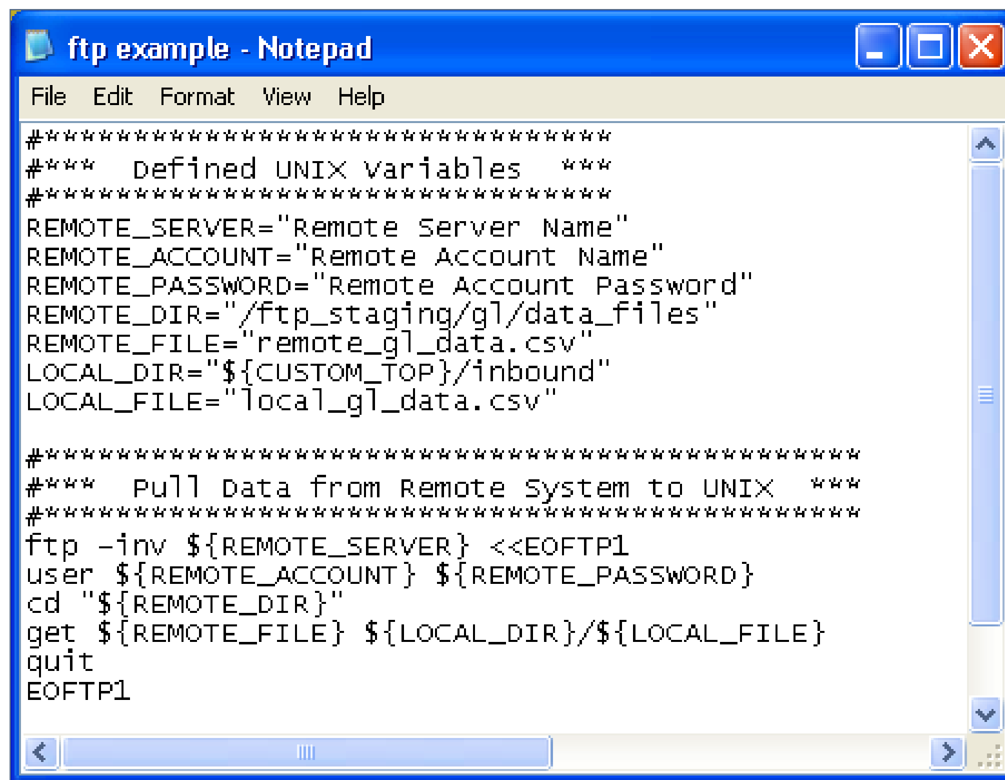
I primarily use UNIX shell programming for sending or retrieving files. Let me show you how I put this into motion. Figure 8 is an example of scripting an FTP process.



unix_option1_setup - Notepad

```
ln -f -s ${FND_TOP}/${APPLBIN}/fndcpesr ${CUSTOM_TOP}/bin/UNIX_SHELL.prog
```

Figure 7: Creating a UNIX Link



ftp example - Notepad

```
#*********************************
#***  Defined UNIX Variables   ***
#*********************************
REMOTE_SERVER="Remote Server Name"
REMOTE_ACCOUNT="Remote Account Name"
REMOTE_PASSWORD="Remote Account Password"
REMOTE_DIR="/ftp_staging/gl/data_files"
REMOTE_FILE="remote_gl_data.csv"
LOCAL_DIR="${CUSTOM_TOP}/inbound"
LOCAL_FILE="local_gl_data.csv"

#*********************************************
#***  Pull Data from Remote System to UNIX  ***
#*********************************************
ftp -inv ${REMOTE_SERVER} <<EOFTP1
user ${REMOTE_ACCOUNT} ${REMOTE_PASSWORD}
cd "${REMOTE_DIR}"
get ${REMOTE_FILE} ${LOCAL_DIR}/${LOCAL_FILE}
quit
EOFTP1
```

Figure 8: FTP

ORAtips © 2006 Klee Associates, Inc.

It is easier to use SQL*Loader for loading data into staging than using utl_file to read a flat file.

To keep things simple for this example, I have defined the UNIX variables prior to actually executing the FTP. The values are bogus. However, you could use this same example to perform an FTP; simply substitute your values for the ones I have provided. Once the UNIX variables are set, you are ready to perform the FTP. The first step is starting the FTP. Since this is within a UNIX shell program, I give a starting and ending header label (EOFTP1). What this says is that everything within EOFTP1 will be part of the FTP process established. Once the process is started, I provide the account to be logged into plus the password associated with the account. Depending on how the account is set up, it may or may not log you into the correct directory. I assume that it does not. In this case, I issue a change directory command to a known directory (called "/ftp_staging/gl/data_files") where the file I wish to pull is located. Once this is done, I pull the data file (called "remote_gl_data.csv") from the remote server to UNIX issuing the "get" command. I also assume that there is a staging directory on UNIX under my custom directory architecture called "inbound". I will place the data file in the custom inbound directory and rename it to "local_gl_data.csv". Once this is completed, the FTP process is done and I quit. Control is now returned to UNIX.

SQL*Loader

The next step with any inbound interface that I deploy is to put the data into a custom staging table in the database. It is possible to read the data directly using the utl_file utility provided by Oracle. However, that requires UNIX directory setups, possibly a bounce of the server. It is a one-time deal, probably requiring support from either your UNIX and/or database administrators. However, I like to keep things simple: the fewer bottlenecks in deployment the better. Also, I find it easier use SQL*Loader for loading data into staging than using utl_file to read a flat file. In our interface, I will assume we are going to use SQL*Loader to load the data from the imported file into a custom staging table. Another benefit is that we can manipulate the data easier using SQL as opposed to editing a file via utl_file.

Figure 9 is a sample of code that can be embedded in a UNIX shell program that executes SQL*Loader to read a data file and load it into a table in Oracle.

Again for simplicity, I have defined the UNIX variables. The exceptions are in the naming of the Log, Bad Data, and Discarded Data files, which can be generated by executing SQL*Loader. In this example, I assume that the developer has put together a SQL*Loader control file called "control_file.ctl". The name of the data file that will be loaded is "local_gl_data.csv" and it is located in a directory called "inbound" under the Custom Directory architecture. From Figure 5 or Figure 6, I use the Request ID that is passed down from the calling concurrent program. I pre-tag the file with "s" concatenated to the Request ID for the Log, Discard, and Bad Data files. By looking at the extensions, I know which file is which. Since the Request ID is attached, I know from which Request ID in the concurrent manager these SQL*Loader files were spawned. Everything is tied together. Simply put, it makes it easier for maintenance.

Executing SQL*Loader from the command prompt involves issuing the "sqlldr" command followed by the Oracle account/password connection string. This is retrieved from values passed from the calling concurrent program (see either Figure 5 or 6). At this point, I tell SQL*Loader the name of the Log report to be generated, the Data file to load, the Control file to read in loading the Data file, the name for the Discarded data file, and the name for the Bad data file.

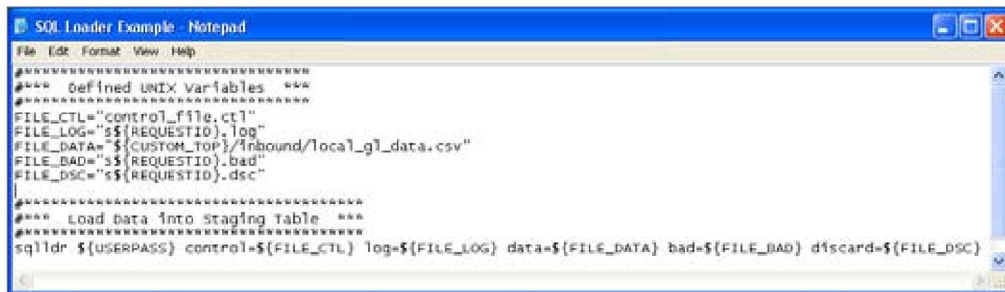


Figure 9: SQL*Loader via UNIX Shell Program

ORAtips © 2006 Klee Associates, Inc.

One important note: SQL*Loader can complete in error, usually due to a formatting issue in the data file. What I do to see if SQL*Loader completed successfully is to determine whether the Bad data file was generated. If the file was generated, an error occurred during SQL*Loader. Since an error occurred, I do not want to continue processing the interface. Hence, I force the interface to terminate. This is accomplished using the "exit" command. I will give a sample of this code under Messaging via E-mail.

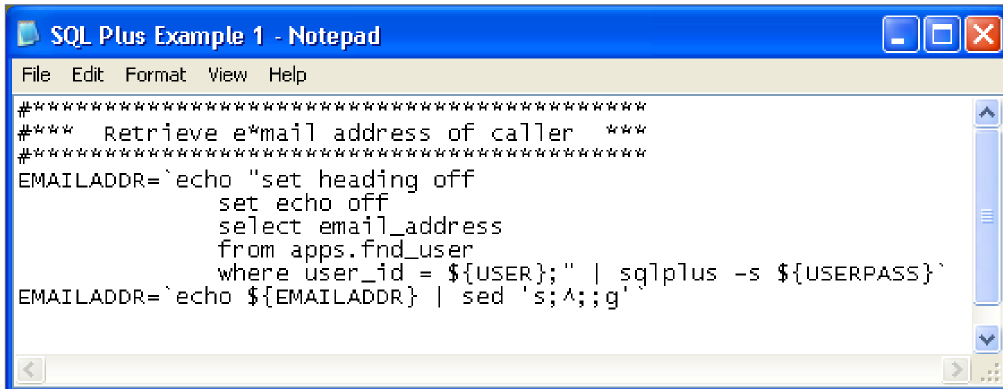


Figure 10: SQL via UNIX Shell Program

SQL*Plus

In this section, I will demonstrate two ways in which SQL*Plus can be used within a UNIX shell program. One way is by directly issuing SQL commands. The other is by executing PL/SQL scripts. We will start be examining direct SQL commands. A sample of code for this method is demonstrated in Figure 10.

In this example, the goal is to retrieve the e-mail address for the caller of the concurrent program. One of the things I like to do with an interface is send messages via an e-mail back to the caller of the program. In this case, the caller (referenced by ${USER}) is a parameter passed to the UNIX shell program via the concurrent program. One of the fields in the AOL table called fnd_user is the email_address. Note, this field must be populated by the system administrator for this activity to work. In any event, let's assume that it is populated for every user who has access to Oracle Applications. Before executing the SQL, I make sure that headers are turned off, plus I do not want the SQL code to be echoed out to the screen. The only thing I want is a single value or the e-mail address. This statement is piped to a SQL*Plus session. Connecting to the session requires an account plus password, which is stored in the UNIX parameter USERPASS. This parameter is populated at the beginning of the UNIX shell program via the concurrent program (see either Figure 5 or 6). The result is stored in the UNIX variable EMAILADDR. However, there might be an unwanted character in the return string, such as indicated by the character "^". I simply replace the occurrence of this character with null. Now, EMAILADDR contains a clean value representing the e-mail address of the caller from the concurrent program.

When embedding SQL commands in UNIX shell programs, do not use tabs in the SQL code section as there is a problem interpreting tabs in SQL code. It may look like I have used tabs, but I only hit the space bar to line everything up. You can use this approach to generate various types of SQL statements.

The other approach to using SQL*Plus in a UNIX shell program is by executing a PL/SQL script. Figure 11 is a code snap-let I have put together to explain this approach.

In this example, I assume that the UNIX variable FILE_PLSQL represents a PL/SQL program name. The variable can also include the path where the PL/SQL program is located. Again, an account and password are required to execute SQL*Plus (USERPASS). For this example, there are seven parameters that are passed to the PL/SQL program. Once the PL/SQL program completes, a parameter is returned,

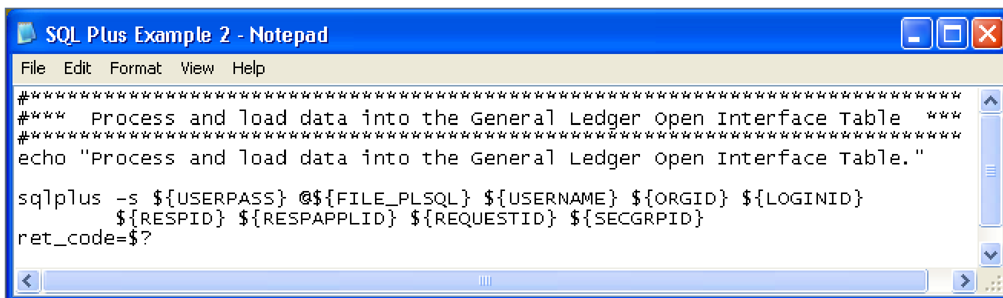
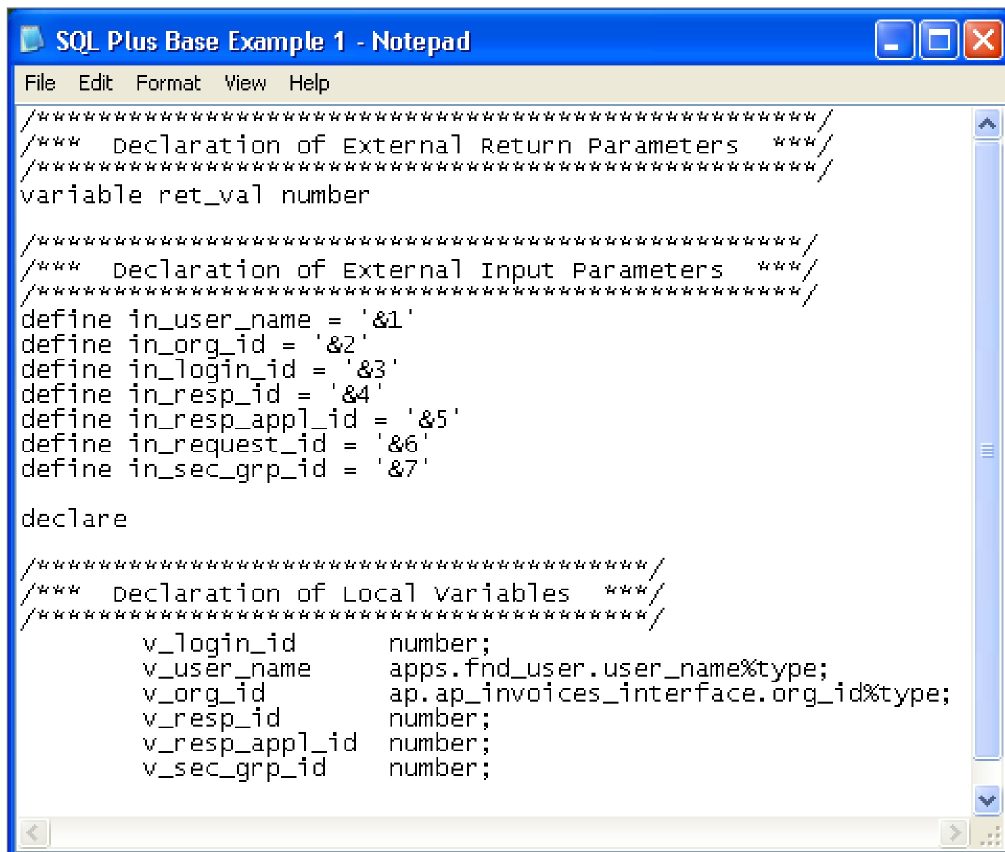


Figure 11: PL/SQL Scripts via UNIX Shell Program

ORAtips © 2006 Klee Associates, Inc.

indicated by "$?". This parameter (known as the return code) can be used to determine if everything programmed in the PL/SQL performed as designed without error. In the UNIX shell program, the value is captured via the variable "ret_code". Let's transition to the PL/SQL program.

The first thing that is performed in the PL/SQL program is the definition of remote and local parameters (see Figure 12). The parameters are gathered and loaded into local variables (see Figure 13). Additionally, I defined one external return parameter. This is done so that when control is returned to UNIX, I can determine if the PL/SQL or any other type of processing within the program performed correctly and without error.



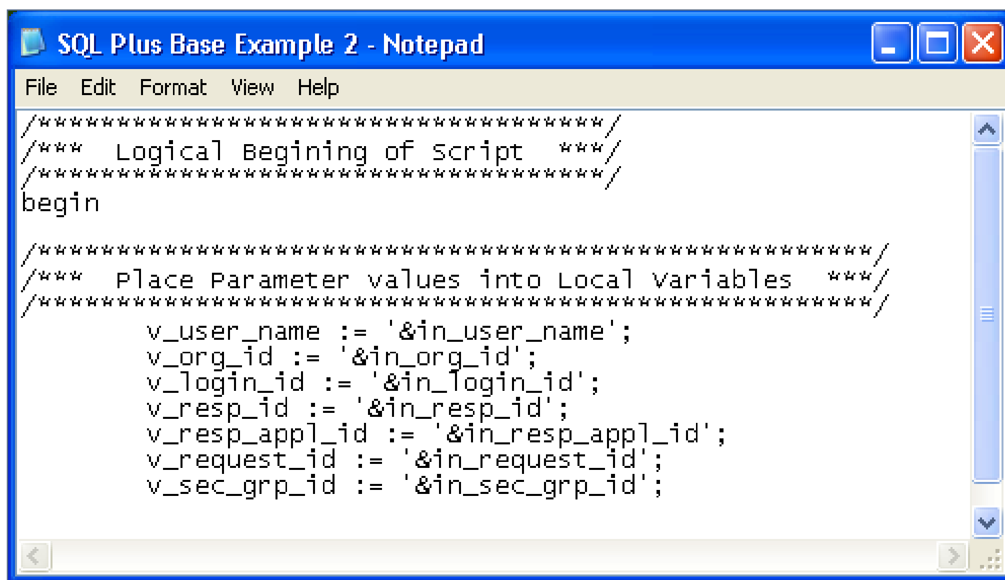

```
/*******************************************************/
/***  Declaration of External Return Parameters   ***/
/*******************************************************/
variable ret_val number

/*******************************************************/
/***  Declaration of External Input Parameters   ***/
/*******************************************************/
define in_user_name = '&1'
define in_org_id = '&2'
define in_login_id = '&3'
define in_resp_id = '&4'
define in_resp_appl_id = '&5'
define in_request_id = '&6'
define in_sec_grp_id = '&7'

declare

/******************************************/
/***  Declaration of Local Variables   ***/
/******************************************/
        v_login_id      number;
        v_user_name     apps.fnd_user.user_name%type;
        v_org_id        ap.ap_invoices_interface.org_id%type;
        v_resp_id       number;
        v_resp_appl_id  number;
        v_sec_grp_id    number;
```

Figure 12: PL/SQL Script - Parameters




```
/*************************************/
/***  Logical Begining of Script   ***/
/*************************************/
begin

/*******************************************************/
/***  Place Parameter values into Local Variables   ***/
/*******************************************************/
        v_user_name := '&in_user_name';
        v_org_id := '&in_org_id';
        v_login_id := '&in_login_id';
        v_resp_id := '&in_resp_id';
        v_resp_appl_id := '&in_resp_appl_id';
        v_request_id := '&in_request_id';
        v_sec_grp_id := '&in_sec_grp_id';
```

Figure 13: PL/SQL Script - Assigning Values

At this point, all input parameters have been defined, data processing can commence, and validation of the data in the staging table is performed. Also, the data loaded is into Oracle's gl_interface table; the Open Interface Table for General Ledger. I will leave this process for another article and a different topic than is being discussed here. Once all processing is complete, control is returned to UNIX. However, there is a remote parameter that is set prior to returning control. That parameter (ret_code) is a numeric indicator. The numeric indicator is retrieved from the SQL*Plus session (see Figure 11) for proper processing of the interface. Setting the return value (via "ret_code") is demonstrated in Figure 14.

Submitting Requests

There are a couple of ways to execute concurrent requests from the background. One is by using an Oracle-supplied program called CONC-

ORAtips © 2006 Klee Associates, Inc.

SUB. The other approach is using the Oracle-supplied package called FND_REQUEST. I have used both approaches for various interfaces.

Executing CONCSUB will return a request ID.

To stick with the denoted design, the concurrent program we will be executing from the background is "Journal Import". I use CONCSUB to execute this job from the background. Figure 15 demonstrates how this is done within a UNIX shell program.

As mentioned previously, I stated that there would be two request IDs that I would need. We know there is one for the concurrent program that executes the interface. The other is from the background call of this separate concurrent program. Executing CONCSUB will return a request ID. Given this, I encapsulate the call within a UNIX variable. The request ID for the background call is the third value in the string returned. Echoing out the return, I can use AWK to capture the third value. Hence, I now know the request ID to the background call of "Journal Import" and report that back to the caller for review.

Another approach to executing concurrent programs from the background is using FND_REQUEST. In this case, I would write code in the UNIX shell program to execute a PL/SQL program via SQL*Plus. Within the PL/SQL script, I can submit the concurrent program using the procedure SUBMIT_REQUEST. In Figure 16 I have provided an example of executing a concurrent program called "Payables Open Interface Import".

I will not dive too deep into this example other than to demonstrate that a concurrent program can be submitted within PL/SQL code. In this case, "Payables Open Interface Import" is a concurrent program owned by Oracle Payables ("SQLAP"). The program associated with this concurrent program is "APXIMPT". Additionally, various parameters are passed to the call. For further study of this utility, please refer to MetaLink or your Developer's User Guide. In any event, it is another approach for executing a concurrent program from the background.

SQL Plus Base Example 3 - Notepad

```
/*****************************************/
/***   No programs with PL/SQL Script   ***/
/*****************************************/
          :ret_val := 0;

/***************************************/
/***   Declaration of Exceptions   ***/
/***************************************/
exception
          when others then
                    dbms_output.put_line (
                              'ERROR: plsql_script.sql; '||
                              'SQL Code = '||to_char(sqlcode)||', '||
                              rtrim(ltrim(sqlerrm))
                    );
                    :ret_val := 1;
                    raise_application_error (
                              -20001,
                              'ERROR: plsql_script.sql; '||
                              'SQL Code = '||to_char(sqlcode)||', '||
                              rtrim(ltrim(sqlerrm))
                    );
end;
/
exit :ret_val;
```

Figure 14: PL/SQL Script - Return Code

concsub example - Notepad

```
#***   Execute Journal Import   ***
#**********************************
echo "Executing Journal Import..."
CP_RESULT=`CONCSUB ${USERPASS} SQLGL "${RESP_NAME}" ${USERNAME} CONCURRENT SQLGL GLLEZL
        ${GROUP_ID} ${SET_OF_BOOKS} ${POST_ERR_TO_SUSP} ${START_DATE} ${END_DATE}
        ${SUMMARY} ${DESC_FLEXFIELD}`
echo ${CP_RESULT}
CP_REQUEST_ID=`echo ${CP_RESULT} | awk '{print $3}'`
```

Figure 15: CONCSUB via a UNIX Shell Program

ORAtips © 2006 Klee Associates, Inc.

fnd_request example - Notepad

File Edit Format View Help

```
v_req_id := fnd_request.submit_request (
          'SQLAP',
          'APXIIMPT',
          'Payables Open Interface Import',
          null,
          false,
          'EMPAP_HUMVP',
          v_batch_num,
          v_batch_num,
          null,
          null,
          to_date('01-'||v_period, 'DD-MON-YY'),
          'Y',
          'N',
          'N',
          'N',
          1000,
          v_user_id,
          v_login_id
);
```

Figure 16: FND_REQUEST Package

EMail Example - Notepad

File Edit Format View Help

```
#*********************************************************
#***  Determine if SQL*Loader completed successfully  ***
#*********************************************************
ls ${FILE_BAD}
if [ $? -ne 0 ];
then
echo "Data file loaded successfully."
else
P_MESSAGE="Problems occurred during SQL*Loader data load for Request ID ${REQUESTID}."
(cat "${P_MESSAGE}" ${FILE_BAD})|mailx -s "ERROR (${INSTANCE}) - ${CONC_PROG}" ${EMAILADDR}
exit 1
fi
```

Figure 17: Messaging via mailx in UNIX Shell Program

Messaging via E-mail

One of my favorite features of using UNIX as a driver for interfaces is that I can customize my messages to the caller based on the status of the interface in question. As promised from the SQL*Plus section, I will demonstrate how I send messages to the caller. The first part you have already seen: Retrieving the e-mail address of the caller. Once this is done, I am at a point of sending the message. This is accomplished using a UNIX utility called mailx. Figure 17 lists the code for sending an e-mail via mailx.

The customization of the message is based on the status of the interface. In this example, I refer you back to the SQL*Loader section of this article. In the interface I have deployed, I do not allow the interface to continue if any bad records were captured. Hence, if the bad data file was created, then I terminate the interface, causing the concurrent program to complete in error. But before I do that, I prepare a message to send to the caller notifying them of the problem.

The message is composed and stored in the UNIX variable P_MESSAGE. The message contains the Request ID for the concurrent program in question. Once this is set up, it is put together with the actual bad data file and piped to mailx. This information will appear in the body of the e-mail. The variable INSTANCE contains the Oracle instance from which the message originated. Since some of the users are also testers, they may have access to more than one instance. I let them know whether this message is coming from a development, test, or production instance. Likewise, in the message header, I notify the caller of which concurrent program generated this message via the variable CONC_PROG. All of this is sent to the calling user identified by the e-mail address stored in the variable EMAILADDR.

This is just one example of sending an e-mail to the caller. You could deploy something similar for problems with the PL/SQL program by examining the return code, or by notifying the user of the request ID generated for the background concurrent program executed. If no problems occurred, I send a success e-mail to the caller.

ORAtips © 2006 Klee Associates, Inc.

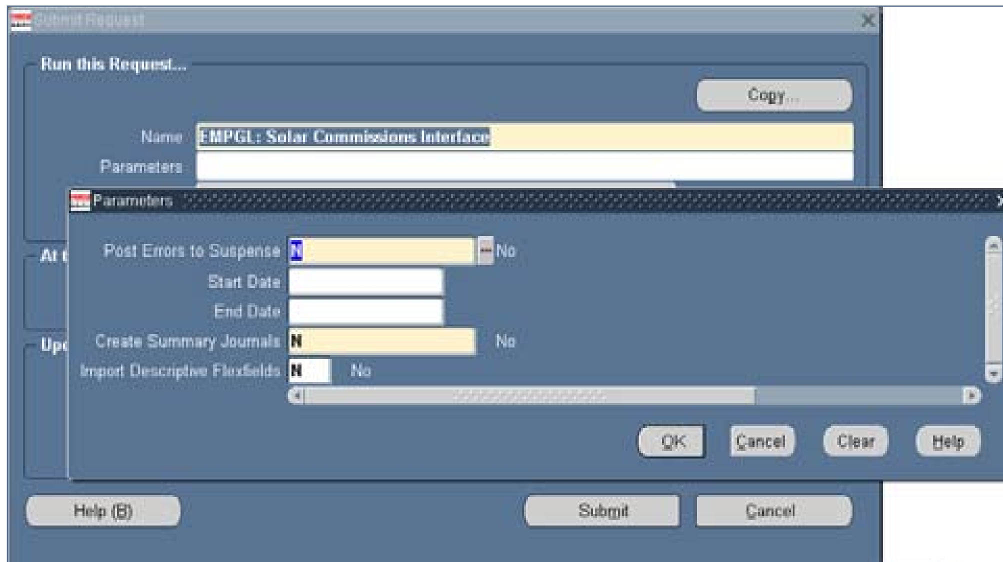


Figure 18: Setting Parameters for Concurrent Program

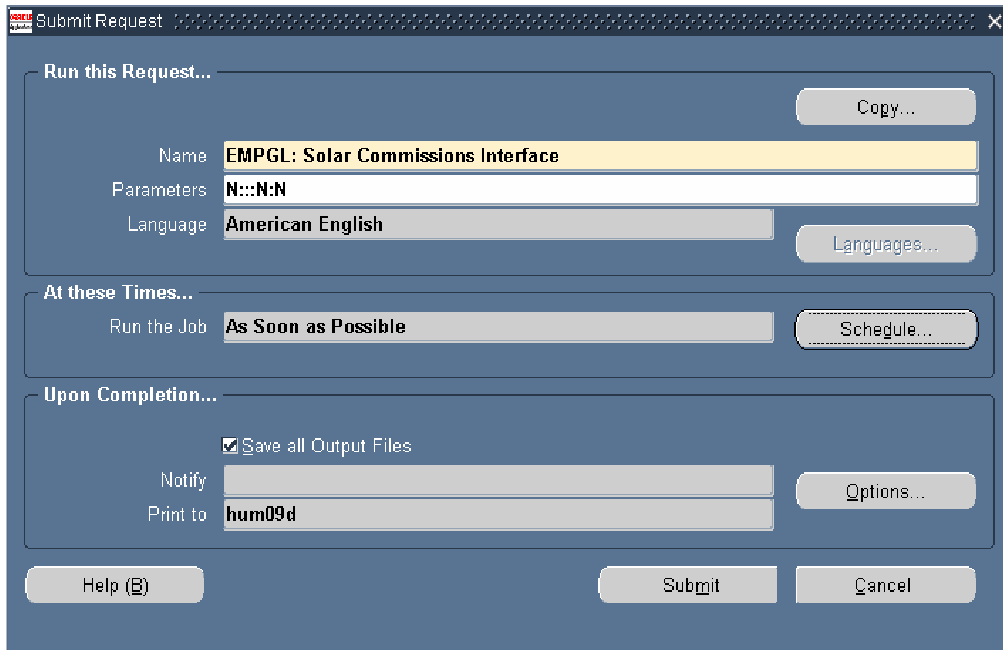


Figure 19: Submit Concurrent Program

Interface in Action

We are now at the final stages of the interface. Most of this article has been showing how to do this from the background. Let me show you how it looks when executed from Oracle Applications. Figures 18 and 19 show the concurrent program used for this demonstration, plus what was used in recording the results for the SWAT analysis.

Once this job is submitted, a concurrent request will be generated. In this case, request ID 264852 is the process for our main interface. This is demonstrated in Figure 20.

Eventually, a sub-process kicked off from the backup will be generated. In this case, Journal Import is the background process kicked off. It is referenced by request ID 264853. Figure 21 demonstrates this.

ORAtips © 2006 Klee Associates, Inc.

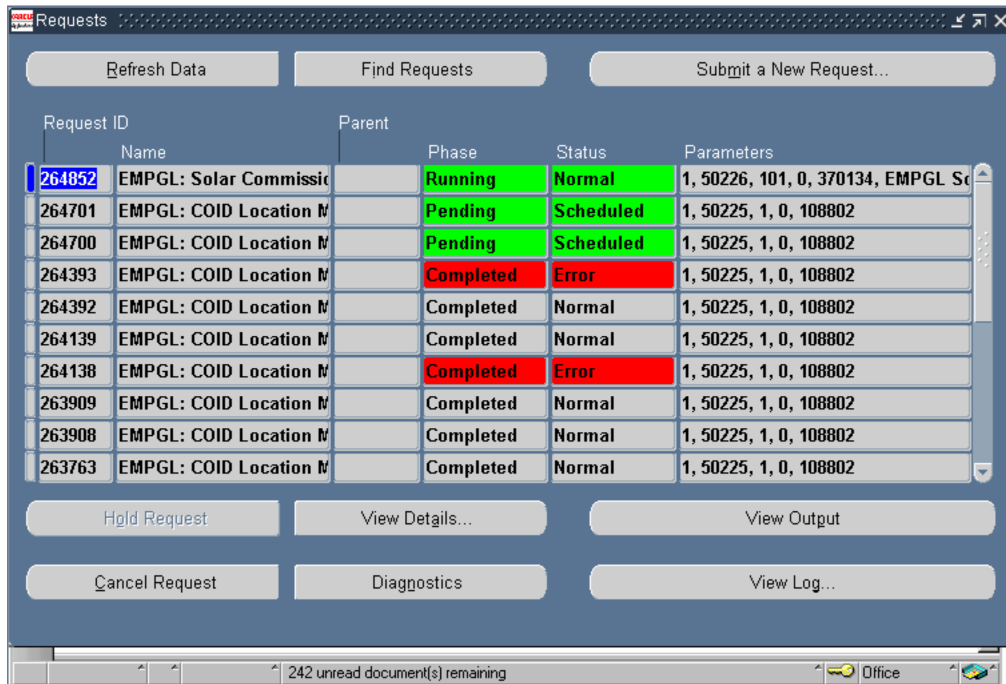


Figure 20: Program Running in Concurrent Manager

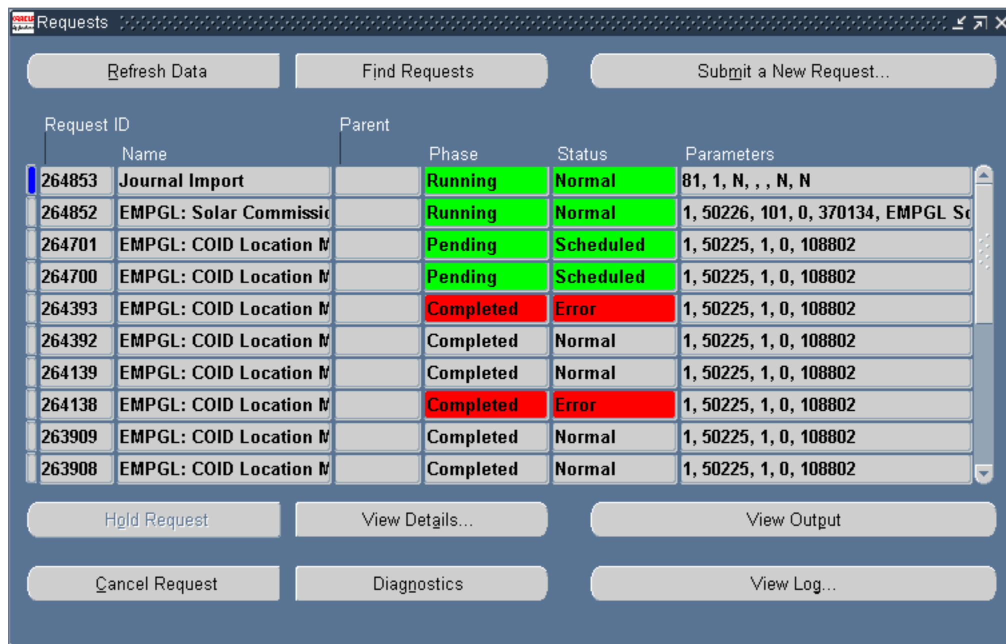


Figure 21: Background Process Spawned

ORAtips © 2006 Klee Associates, Inc.

ORAtips

On Development

If everything works perfectly, both processes should complete successfully (see Figure 22).

The final stage is the completion e-mail. Below is the message generated from the interface (see Figure 23). In this sample message, a status (either SUCCESS or ERROR) is indicated along with the name of the job it pertains to. Additionally, it records the instance that the process happened in along with the corresponding Request ID. In this case there are background processes (Journal Import) that were submitted. The Request ID for this job is also included so the associate can review the appropriate report in the concurrent manager.

Conclusion

Due to the speed, efficiency, and reduction in request IDs to maintain, I try to approach any project that requires an interface by using a single concurrent program. This does not mean that a single concurrent program is the answer for every situation. You may wish for your users to have control of the processes (or stages) and how they run. If this is the case, the request set may be your best option. However, if you are running interfaces that process large amounts of data or if you have on-demand interfaces that process batch

Requests

Refresh Data | Find Requests | Submit a New Request...

Request ID	Name	Parent	Phase	Status	Parameters
264853	Journal Import		Completed	Normal	81, 1, N, , , N, N
264852	EMPGL: Solar Commissic		Completed	Normal	1, 50226, 101, 0, 370134, EMPGL Sc
264701	EMPGL: COID Location M		Pending	Scheduled	1, 50225, 1, 0, 108802
264700	EMPGL: COID Location M		Pending	Scheduled	1, 50225, 1, 0, 108802
264393	EMPGL: COID Location M		Completed	Error	1, 50225, 1, 0, 108802
264392	EMPGL: COID Location M		Completed	Normal	1, 50225, 1, 0, 108802
264139	EMPGL: COID Location M		Completed	Normal	1, 50225, 1, 0, 108802
264138	EMPGL: COID Location M		Completed	Error	1, 50225, 1, 0, 108802
263909	EMPGL: COID Location M		Completed	Normal	1, 50225, 1, 0, 108802
263908	EMPGL: COID Location M		Completed	Normal	1, 50225, 1, 0, 108802

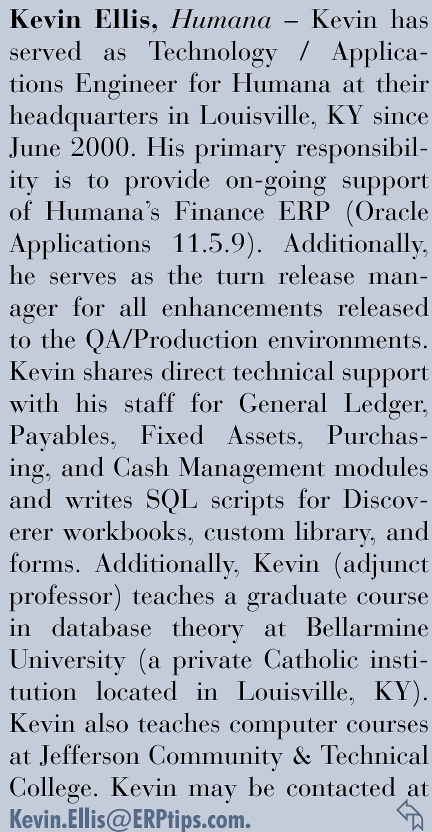
Hold Request | View Details... | View Output
Cancel Request | Diagnostics | View Log...

Figure 22: Jobs Complete Successfully

Figure 23: Body of the E-mail

ORAtips © 2006 Klee Associates, Inc.

data, my suggestion is to explore the idea of deploying an interface using a single concurrent program driven by a UNIX shell program. It will save you time in analyzing problems, callers of the program are immediately notified of the problem, and the process runs faster.


Kevin Ellis, *Humana* – Kevin has served as Technology / Applications Engineer for Humana at their headquarters in Louisville, KY since June 2000. His primary responsibility is to provide on-going support of Humana's Finance ERP (Oracle Applications 11.5.9). Additionally, he serves as the turn release manager for all enhancements released to the QA/Production environments. Kevin shares direct technical support with his staff for General Ledger, Payables, Fixed Assets, Purchasing, and Cash Management modules and writes SQL scripts for Discoverer workbooks, custom library, and forms. Additionally, Kevin (adjunct professor) teaches a graduate course in database theory at Bellarmine University (a private Catholic institution located in Louisville, KY). Kevin also teaches computer courses at Jefferson Community & Technical College. Kevin may be contacted at **Kevin.Ellis@ERPtips.com.**

ORAtips © 2006 Klee Associates, Inc.

ORAtips Journal

The information on our website and in our publications is the copyrighted work of Klee Associates, Inc. and is owned by Klee Associates, Inc. NO WARRANTY: This documentation is delivered as is, and Klee Associates, Inc. makes no warranty as to its accuracy or use. Any use of this documentation is at the risk of the user. Although we make every good faith effort to ensure accuracy, this document may include technical or other inaccuracies or typographical errors. Klee Associates, Inc. reserves the right to make changes without prior notice. NO AFFILIATION: Klee Associates, Inc. and this publication are not affiliated with or endorsed by Oracle Corporation. Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Klee Associates, Inc. is a member of the Oracle Partner Network

This article was originally published by Klee Associates, Inc., publishers of JDEtips and SAPtips. For training, consulting, and articles on JD Edwards or SAP, please visit our websites: www.JDEtips.com and www.SAPtips.com.

ORAtips © 2006 Klee Associates, Inc.