Define your job Procedure\_JB to accept arguments. Then use dbms\_scheduler.set\_job\_argument\_value to define the value of the arguments you want to pass into the program your job is going to run. An example (taken from <https://forums.oracle.com/forums/thread.jspa?threadID=483135>)

-- create a stored procedure with two arguments

create or replace procedure myproc (arg1 in varchar2, arg2 in varchar2)

is BEGIN null; END;

/

-- create a program with two arguments and define both

begin

dbms\_scheduler.create\_program

(

program\_name=>'myprog',

program\_action=>'myproc',

program\_type=>'STORED\_PROCEDURE',

number\_of\_arguments=>2, enabled=>FALSE

) ;

dbms\_scheduler.DEFINE\_PROGRAM\_ARGUMENT(

program\_name=>'myprog',

argument\_position=>1,

argument\_type=>'VARCHAR2',

DEFAULT\_VALUE=>'13');

dbms\_scheduler.DEFINE\_PROGRAM\_ARGUMENT(

program\_name=>'myprog',

argument\_position=>2,

argument\_type=>'VARCHAR2');

dbms\_scheduler.enable('myprog');

end;

/

-- create a job pointing to a program and set both argument values

begin

dbms\_scheduler.create\_job('myjob',program\_name=>'myprog');

dbms\_scheduler.set\_job\_argument\_value('myjob',1,'first arg');

dbms\_scheduler.set\_job\_argument\_value('myjob',2,'second arg');

dbms\_scheduler.enable('myjob');

end;

/

Use PLSQL\_BLOCK instead:

begin

dbms\_scheduler.create\_job (

job\_name => 'myjob',

job\_type => 'PLSQL\_BLOCK',

job\_action => 'BEGIN myproc(''first arg'',''second arg''); END;',

start\_date => sysdate,

repeat\_interval => 'FREQ=HOURLY',

enabled => true

);

end;

/

as the error states, create a program first, then the job on that.

dbms\_scheduler.create\_program(program\_name => 'YOUR\_PROGRAM',

program\_type => 'STORED\_PROCEDURE',

program\_action => 'my\_test\_proc',

number\_of\_arguments => 2,

enabled => false,

comments => 'Comments you want');

dbms\_scheduler.define\_program\_argument(program\_name => 'YOUR\_PROGRAM',

argument\_name => 'param1',

argument\_position => 1,

argument\_type => 'VARCHAR2',

default\_value => '');

..etc, do for all 3.

dbms\_scheduler.enable (name => 'YOUR\_PROGRAM');

dbms\_scheduler.create\_job(job\_name => 'my\_test\_job',

program\_name => 'YOUR\_PROGRAM',

start\_date => systimestamp,

end\_date => null,

...

dbms\_scheduler.set\_job\_argument\_value(job\_name => 'my\_test\_job',

argument\_position => 1,

argument\_value => 'value');

You should first create the job, then define the arguments, and then run it. When you create it, set the enabled atribute to false, so it won't run yet:

dbms\_scheduler.create\_job(job\_name => 'parse\_job',

program\_name => 'PARSE\_PROGRAM',

start\_date => systimestamp,

enabled => false );

Then pass the arguments to the job:

dbms\_scheduler.set\_job\_argument\_value(job\_name => 'parse\_job',

argument\_position => 1,

argument\_value => 1);

Then enable it with a call:

dbms\_scheduler.enable('parse\_job');

begin

sys.dbms\_scheduler.create\_job(job\_name => 'Your Job name',

job\_type => 'PLSQL\_BLOCK',

job\_action => 'begin schema.packagename.procedurename(parametername=> parametervalue); end;',

start\_date => to\_date('01-01-2015 00:00:00', 'dd-mm-yyyy hh24:mi:ss'),

repeat\_interval => 'Freq=Monthly;Interval=1',

end\_date => to\_date(null),

job\_class => 'DEFAULT\_JOB\_CLASS',

enabled => false,

auto\_drop => true,

comments => 'Job comment');

end;

You may use a FOR LOOP by setting first the start job number and end job numbers.

DECLARE

v\_start\_job\_number INTEGER := 8;

v\_end\_job\_number INTEGER := 17;

BEGIN

FOR v\_job\_number IN v\_start\_job\_number .. v\_end\_job\_number

LOOP

sys.DBMS\_SCHEDULER.create\_job (

job\_name => 'WEB.TEMP\_COMPILE\_JOB\_' || v\_job\_number,

job\_type => 'PLSQL\_BLOCK',

job\_action => 'BEGIN

WEB.PAGE\_REFRESH.compile\_page('

|| v\_job\_number

|| ',TRUE);

END;',

start\_date => FROM\_TZ (SYS\_EXTRACT\_UTC (LOCALTIMESTAMP), 'UTC'),

job\_class => 'DEFAULT\_JOB\_CLASS',

comments => 'refresh',

auto\_drop => TRUE,

enabled => TRUE);

END LOOP;

END;

=================

Dbms\_scheduler allows to create program definition with arguments. One problem is that boolean type is not support but it's not a big obstacle.

Example:

create table log\_do\_somthing(p1 number, p2 varchar2(10));

create or replace procedure do\_somthing(p1 number, p2 varchar2) is

begin

insert into log\_do\_somthing values ( p1,p2);

commit;

end;

Dummy structure

begin

dbms\_scheduler.create\_program

(

program\_name=>'do\_somthing\_prog',

program\_action=>'do\_somthing',

program\_type=>'STORED\_PROCEDURE',

number\_of\_arguments=>2, enabled=>FALSE

) ;

dbms\_scheduler.DEFINE\_PROGRAM\_ARGUMENT(

program\_name=>'do\_somthing\_prog',

argument\_position=>1,

argument\_type=>'NUMBER');

dbms\_scheduler.DEFINE\_PROGRAM\_ARGUMENT(

program\_name=>'do\_somthing\_prog',

argument\_position=>2,

argument\_type=>'VARCHAR2');

dbms\_scheduler.enable('do\_somthing\_prog');

end;

/

Creating job using defined program and setting parameters.

declare

job\_name varchar2(100);

begin

for i in 1 .. 10 loop

job\_name := DBMS\_SCHEDULER.GENERATE\_JOB\_NAME('do\_somthin\_proefix');

dbms\_scheduler.create\_job(job\_name,program\_name=>'do\_somthing\_prog');

dbms\_scheduler.set\_job\_argument\_value(job\_name,1,to\_char(i));

dbms\_scheduler.set\_job\_argument\_value(job\_name,2,'TURE');

dbms\_scheduler.enable(job\_name);

end loop;

end;

/

Naming convention in the scheduler package is straightforward and meaningful. But if you need more information description of all methods is here <https://docs.oracle.com/database/121/ARPLS/d_sched.htm#ARPLS72235>

==============

<https://oracle-base.com/articles/10g/scheduler-10g>

================

***7.12******DBMS\_SCHEDULER***

The DBMS\_SCHEDULER package provides a way to create and manage Oracle-styled jobs, programs and job schedules.

**Table 7.7.2 DBMS\_SCHEDULER Functions and Procedures**

|  |  |  |
| --- | --- | --- |
| **Function/Procedure** | **Return Type** | **Description** |
| CREATE\_JOB(*job\_name*, *job\_type*, *job\_action*, *number\_of\_arguments*, *start\_date*, *repeat\_interval*, *end\_date*, *job\_class*, *enabled*, *auto\_drop*, *comments*) | n/a | Use the first form of the CREATE\_JOB procedure to create a job, specifying program and schedule details by means of parameters. |
| CREATE\_JOB(*job\_name*, *program\_name*, *schedule\_name*, *job\_class*, *enabled*, *auto\_drop*, *comments*) | n/a | Use the second form of CREATE\_JOB to create a job that uses a named program and named schedule. |
| CREATE\_PROGRAM(*program\_name*, *program\_type*, *program\_action*, *number\_of\_arguments*, *enabled*, *comments*) | n/a | Use CREATE\_PROGRAM to create a program. |
| CREATE\_SCHEDULE( *schedule\_name*, *start\_date*, *repeat\_interval*, *end\_date*, *comments*) | n/a | Use the CREATE\_SCHEDULE procedure to create a schedule. |
| DEFINE\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_position*, *argument\_name*, *argument\_type*, *default\_value*, *out\_argument*) | n/a | Use the first form of the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument that has a default value. |
| DEFINE\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_position*, *argument\_name*, *argument\_type*, *out\_argument*) | n/a | Use the first form of the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument that does not have a default value. |
| DISABLE(*name*, *force*, *commit\_semantics*) | n/a | Use the DISABLE procedure to disable a job or program. |
| DROP\_JOB(*job\_name*, *force*, *defer*, *commit\_semantics*) | n/a | Use the DROP\_JOB procedure to drop a job. |
| DROP\_PROGRAM(*program\_name*, *force*) | n/a | Use the DROP\_PROGRAM procedure to drop a program. |
| DROP\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_position*) | n/a | Use the first form of DROP\_PROGRAM\_ARGUMENT to drop a program argument by specifying the argument position. |
| DROP\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_name*) | n/a | Use the second form of DROP\_PROGRAM\_ARGUMENT to drop a program argument by specifying the argument name. |
| DROP\_SCHEDULE(*schedule\_name*, *force*) | n/a | Use the DROP SCHEDULE procedure to drop a schedule. |
| ENABLE(*name*, *commit\_semantics*) | n/a | Use the ENABLE command to enable a program or job. |
| EVALUATE\_CALENDAR\_STRING( *calendar\_string*, *start\_date*, *return\_date\_after*, *next\_run\_date*) | n/a | Use EVALUATE\_CALENDAR\_STRING to review the execution date described by a user-defined calendar schedule. |
| RUN\_JOB(*job\_name*, *use\_current\_session*, *manually*) | n/a | Use the RUN\_JOB procedure to execute a job immediately. |
| SET\_JOB\_ARGUMENT\_VALUE( *job\_name*, *argument\_position*, *argument\_value*) | n/a | Use the first form of SET\_JOB\_ARGUMENT value to set the value of a job argument described by the argument's position. |
| SET\_JOB\_ARGUMENT\_VALUE( *job\_name*, *argument\_name*, *argument\_value*) | n/a | Use the second form of SET\_JOB\_ARGUMENT value to set the value of a job argument described by the argument's name. |

Postgres Plus Advanced Server's implementation of DBMS\_SCHEDULER is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The DBMS\_SCHEDULER package is dependent on the pgAgent service; you must have a pgAgent service installed and running on your server before using DBMS\_SCHEDULER.

Before using DBMS\_SCHEDULER, a database superuser must create the catalog tables in which the DBMS\_SCHEDULER programs, schedules and jobs are stored. Use the psql client to connect to the database, and invoke the command:

CREATE EXTENSION dbms\_scheduler;

By default, the dbms\_scheduler extension resides in the contrib/dbms\_scheduler\_ext subdirectory (under the Advanced Server installation).

Note that after creating the DBMS\_SCHEDULER tables, only a superuser will be able to perform a dump or reload of the database.

**7.12.1****Using Calendar Syntax to Specify a Repeating Interval**

The CREATE\_JOB and CREATE\_SCHEDULE procedures use Oracle-styled calendar syntax to define the interval with which a job or schedule is repeated. You should provide the scheduling information in the *repeat\_interval* parameter of each procedure.

*repeat\_interval* is a value (or series of values) that define the interval between the executions of the scheduled job. Each value is composed of a token, followed by an equal sign, followed by the unit (or units) on which the schedule will execute. Multiple token values must be separated by a semi-colon (;).

For example, the following value:

FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;BYMINUTE=45

Defines a schedule that is executed each weeknight at 5:45.

The token types and syntax described in the table below are supported by Advanced Server:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Token type** | **Syntax** | **Valid Values** | | |
| FREQ | FREQ=*predefined\_interval* | Where *predefined\_interval* is one of the following: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY. The SECONDLY keyword is not supported. | | |
| BYMONTH | BYMONTH=*month*(, *month*)... | Where *month* is the three-letter abbreviation of the month name: JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC | | |
| BYMONTH | BYMONTH=*month*(, *month*)... | Where *month* is the numeric value representing the month: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | |
| BYMONTHDAY | BYMONTHDAY=*day\_of\_month* | Where *day\_of\_month* is a value from 1 through 31 | | |
| BYDAY | BYDAY=*weekday* | Where *weekday* is a three-letter abbreviation or single-digit value representing the day of the week. | | |
| Monday | MON | 1 |
| Tuesday | TUE | 2 |
| Wednesday | WED | 3 |
| Thursday | THU | 4 |
| Friday | FRI | 5 |
| Saturday | SAT | 6 |
| Sunday | SUN | 7 |
| BYDATE | BYDATE=*date*(, *date*)... | Where *date*is *YYYYMMDD*.  YYYY is a four-digit year representation of the year, MM is a two-digit representation of the month, and DD is a two-digit day representation of the day. | | |
| BYDATE | BYDATE=*date*(, *date*)... | Where *date*is *MMDD*.  MM is a two-digit representation of the month, and DD is a two-digit day representation of the day | | |
| BYHOUR | BYHOUR=*hour* | Where *hour*is a value from 0 through 23. | | |
| BYMINUTE | BYMINUTE=*minute* | Where *minute*is a value from 0 through 59. | | |

**7.12.2****CREATE\_JOB**

Use the CREATE\_JOB procedure to create a job. The procedure comes in two forms; the first form of the procedure specifies a schedule within the job definition, as well as a job action that will be invoked when the job executes:

CREATE\_JOB(  
 *job*\_*name* IN VARCHAR2,   
 *job\_type* IN VARCHAR2,   
 *job\_action* IN VARCHAR2,   
 *number\_of\_arguments* IN PLS\_INTEGER DEFAULT 0,   
 *start\_date* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 *repeat\_interval* IN VARCHAR2 DEFAULT NULL,   
 *end\_date* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 *job\_class* IN VARCHAR2 DEFAULT 'DEFAULT\_JOB\_CLASS',  
 *enabled* IN BOOLEAN DEFAULT FALSE,  
 *auto\_drop* IN BOOLEAN DEFAULT TRUE,  
 *comments* IN VARCHAR2 DEFAULT NULL)

The second form uses a job schedule to specify the schedule on which the job will execute, and specifies the name of a program that will execute when the job runs:

CREATE\_JOB(  
 *job\_name* IN VARCHAR2,  
 *program\_name* IN VARCHAR2,  
 *schedule\_name* IN VARCHAR2,  
 *job\_class* IN VARCHAR2 DEFAULT 'DEFAULT\_JOB\_CLASS',  
 *enabled* IN BOOLEAN DEFAULT FALSE,  
 *auto\_drop* IN BOOLEAN DEFAULT TRUE,  
 *comments* IN VARCHAR2 DEFAULT NULL)

**Parameters**

*job*\_*name*

*job\_name* specifies the optionally schema-qualified name of the job being created.

*job\_type*

*job\_type* specifies the type of job. The current implementation of CREATE\_JOB supports a job type of PLSQL\_BLOCK or STORED\_PROCEDURE.

*job\_action*

If *job\_type* is PLSQL\_BLOCK, *job\_action* specifies the content of the PL/SQL block that will be invoked when the job executes. The block must be terminated with a semi-colon (;).

If *job\_type* is STORED\_PROCEDURE, *job\_action* specifies the optionally schema-qualified name of the procedure.

*number\_of\_arguments*

*number\_of\_arguments* is an INTEGER value that specifies the number of arguments expected by the job. The default is 0.

*start\_date*

*start\_date* is a TIMESTAMP WITH TIME ZONE value that specifies the first time that the job is scheduled to execute. The default value is NULL, indicating that the job should be scheduled to execute when the job is enabled.

*repeat\_interval*

*repeat\_interval* is a VARCHAR2 value that specifies how often the job will repeat. If a *repeat\_interval* is not specified, the job will execute only once. The default value is NULL.

For information about defining a repeating schedule for a job, see [Section 7.12.1](https://www.enterprisedb.com/edb-docs/d/edb-postgres-advanced-server/user-guides/database-compatibility-for-oracle-developers-guide/9.4/Database_Compatibility_for_Oracle_Developers_Guide.1.188.html#pID0E0AMJ0HA).

*end\_date*

*end\_date* is a TIMESTAMP WITH TIME ZONE value that specifies a time after which the job will no longer execute. If a date is specified, the *end\_date* must be after *start\_date*. The default value is NULL.

Please note that if an *end\_date* is not specified and a *repeat\_interval* is specified, the job will repeat indefinitely until it is disabled.

*program\_name*

*program\_name* is the name of a program that will be executed by the job.

*schedule\_name*

*schedule\_name* is the name of the schedule associated with the job.

*job\_class*

*job\_class* is accepted for compatibility and ignored.

*enabled*

*enabled* is a BOOLEAN value that specifies if the job is enabled when created. By default, a job is created in a disabled state, with *enabled* set to FALSE. To enable a job, specify a value of TRUE when creating the job, or enable the job with the DBMS\_SCHEDULER.ENABLE procedure.

*auto\_drop*

The *auto\_drop* parameter is accepted for compatibility and is ignored. By default, a job's status will be changed to DISABLED after the time specified in *end\_date*.

*comments*

Use the *comments* parameter to specify a comment about the job.

**Example**

The following example demonstrates a call to the CREATE\_JOB procedure:

EXEC

DBMS\_SCHEDULER.CREATE\_JOB (

job\_name => 'update\_log',

job\_type => 'PLSQL\_BLOCK',

job\_action => 'BEGIN INSERT INTO my\_log VALUES(current\_timestamp);  
 END;',

start\_date => '01-JUN-15 09:00:00.000000',

repeat\_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',

end\_date => NULL,

enabled => TRUE,

comments => 'This job adds a row to the my\_log table.');

The code fragment creates a job named update\_log that executes each weeknight at 5:00. The job executes a PL/SQL block that inserts the current timestamp into a logfile (my\_log). Since no end\_date is specified, the job will execute until it is disabled by the DBMS\_SCHEDULER.DISABLE procedure.

**7.12.3****CREATE\_PROGRAM**

Use the CREATE\_PROGRAM procedure to create a DBMS\_SCHEDULER program. The signature is:

CREATE\_PROGRAM(  
 *program\_name* IN VARCHAR2,  
 *program\_type* IN VARCHAR2,  
 *program\_action* IN VARCHAR2, *number\_of\_arguments* IN PLS\_INTEGER DEFAULT 0,  
 *enabled* IN BOOLEAN DEFAULT FALSE,  
 *comments* IN VARCHAR2 DEFAULT NULL)

**Parameters**

*program\_name*

*program\_name* specifies the name of the program that is being created.

*program\_type*

*program\_type* specifies the type of program. The current implementation of CREATE\_PROGRAM supports a *program\_type* of PLSQL\_BLOCK or PROCEDURE.

*program\_action*

If *program\_type* is PLSQL\_BLOCK, *program\_action* contains the PL/SQL block that will execute when the program is invoked. The PL/SQL block must be terminated with a semi-colon (;).

If *program\_type* is PROCEDURE, *program\_action* contains the name of the stored procedure.

*number\_of\_arguments*

If *program\_type* is PLSQL\_BLOCK, this argument is ignored.

If *program*\_*type* is PROCEDURE, *number\_of\_arguments*specifies the number of arguments required by the procedure. The default value is 0.

*enabled*

*enabled* specifies if the program is created enabled or disabled:

|  |  |
| --- | --- |
| • | If *enabled* is TRUE, the program is created enabled. |
| • | If *enabled* is FALSE, the program is created disabled; use the DBMS\_SCHEDULER.ENABLE program to enable a disabled program. |

The default value is FALSE.

*comments*

Use the *comments* parameter to specify a comment about the program; by default, this parameter is NULL.

**Example**

The following call to the CREATE\_PROGRAM procedure creates a program named update\_log:

EXEC

DBMS\_SCHEDULER.CREATE\_PROGRAM (  
 program\_name => 'update\_log',

program\_type => 'PLSQL\_BLOCK',

program\_action => 'BEGIN INSERT INTO my\_log VALUES(current\_timestamp);  
 END;',

enabled => TRUE,

comment => 'This program adds a row to the my\_log table.');

update\_log is a PL/SQL block that adds a row containing the current date and time to the my\_log table. The program will be enabled when the CREATE\_PROGRAM procedure executes.

**7.12.4****CREATE\_SCHEDULE**

Use the CREATE\_SCHEDULE procedure to create a job schedule. The signature of the CREATE\_SCHEDULE procedure is:

CREATE\_SCHEDULE(  
 *schedule\_name* IN VARCHAR2,  
 *start\_date* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 *repeat\_interval* IN VARCHAR2,  
 *end\_date* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 *comments* IN VARCHAR2 DEFAULT NULL)

**Parameters**

*schedule\_name*

*schedule\_name* specifies the name of the schedule.

*start\_date*

*start\_date* is a TIMESTAMP WITH TIME ZONE value that specifies the date and time that the schedule is eligible to execute. If a *start\_date* is not specified, the date that the job is enabled is used as the *start\_date*. By default, *start\_date* is NULL.

*repeat\_interval*

*repeat\_interval* is a VARCHAR2 value that specifies how often the job will repeat. If a *repeat\_interval* is not specified, the job will execute only once, on the date specified by *start\_date*.

For information about defining a repeating schedule for a job, see [Section 7.12.1](https://www.enterprisedb.com/edb-docs/d/edb-postgres-advanced-server/user-guides/database-compatibility-for-oracle-developers-guide/9.4/Database_Compatibility_for_Oracle_Developers_Guide.1.188.html#pID0E0AMJ0HA).

Please note: you must provide a value for either *start\_date* or *repeat\_interval*; if both *start\_date* and *repeat\_interval* are NULL, the server will return an error.

*end\_date* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL

*end\_date* is a TIMESTAMP WITH TIME ZONE value that specifies a time after which the schedule will no longer execute. If a date is specified, the *end\_date* must be after the *start\_date*. The default value is NULL.

Please note that if a *repeat\_interval* is specified and an *end\_date* is not specified, the schedule will repeat indefinitely until it is disabled.

*comments* IN VARCHAR2 DEFAULT NULL)

Use the *comments* parameter to specify a comment about the schedule; by default, this parameter is NULL.

**Example**

The following code fragment calls CREATE\_SCHEDULE to create a schedule named weeknights\_at\_5:

EXEC

DBMS\_SCHEDULER.CREATE\_SCHEDULE (

schedule\_name => 'weeknights\_at\_5',

start\_date => '01-JUN-13 09:00:00.000000'

repeat\_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',

comments => 'This schedule executes each weeknight at 5:00');

The schedule executes each weeknight, at 5:00 pm, effective after June 1, 2013. Since no end\_date is specified, the schedule will execute indefinitely until it is disabled with DBMS\_SCHEDULER.DISABLE.

**7.12.5****DEFINE\_PROGRAM\_ARGUMENT**

Use the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument. The DEFINE\_PROGRAM\_ARGUMENT procedure comes in two forms; the first form defines an argument with a default value:

DEFINE\_PROGRAM\_ARGUMENT(  
 *program\_name* IN VARCHAR2,  
 *argument\_position* IN PLS\_INTEGER,  
 *argument\_name* IN VARCHAR2 DEFAULT NULL,  
 *argument\_type* IN VARCHAR2,  
 *default\_value* IN VARCHAR2,  
 *out\_argument* IN BOOLEAN DEFAULT FALSE)

The second form defines an argument without a default value:

DEFINE\_PROGRAM\_ARGUMENT(  
 *program\_name* IN VARCHAR2,  
 *argument\_position* IN PLS\_INTEGER,  
 *argument\_name* IN VARCHAR2 DEFAULT NULL,  
 *argument\_type* IN VARCHAR2,  
 *out\_argument* IN BOOLEAN DEFAULT FALSE)

**Parameters**

*program\_name*

*program\_name* is the name of the program to which the arguments belong.

*argument\_position*

*argument\_position* specifies the position of the argument as it is passed to the program.

*argument\_name*

*argument\_name* specifies the optional name of the argument. By default, *argument\_name* is NULL.

*argument\_type* IN VARCHAR2

*argument\_type* specifies the data type of the argument.

*default\_value*

*default\_value* specifies the default value assigned to the argument. *default\_value* will be overridden by a value specified by the job when the job executes.

*out\_argument* IN BOOLEAN DEFAULT FALSE

*out\_argument* is not currently used; if specified, the value must be FALSE.

**Example**

The following code fragment uses the DEFINE\_PROGRAM\_ARGUMENT procedure to define the first and second arguments in a program named add\_emp:

EXEC

DBMS\_SCHEDULER.DEFINE\_PROGRAM\_ARGUMENT(

program\_name => 'add\_emp',

argument\_position => 1,

argument\_name => 'dept\_no',

argument\_type => 'INTEGER,

default\_value => '20');

EXEC

DBMS\_SCHEDULER.DEFINE\_PROGRAM\_ARGUMENT(

program\_name => 'add\_emp',

argument\_position => 2,

argument\_name => 'emp\_name',

argument\_type => 'VARCHAR2');

The first argument is an INTEGER value named dept\_no that has a default value of 20. The second argument is a VARCHAR2 value named emp\_name; the second argument does not have a default value.

**7.12.6****DISABLE**

Use the DISABLE procedure to disable a program or a job. The signature of the DISABLE procedure is:

DISABLE(  
 *name* IN VARCHAR2,  
 *force* IN BOOLEAN DEFAULT FALSE,  
 *commit\_semantics* IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

**Parameters**

*name*

*name* specifies the name of the program or job that is being disabled.

*force*

*force* is accepted for compatibility, and ignored.

*commit\_semantics*

*commit\_semantics* instructs the server how to handle an error encountered while disabling a program or job. By default, *commit\_semantics* is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error. Any programs or jobs that were successfully disabled prior to the error will be committed to disk.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

**Example**

The following call to the DISABLE procedure disables a program named update\_emp:

DBMS\_SCHEDULER.DISABLE('update\_emp');

**7.12.7****DROP\_JOB**

Use the DROP\_JOB procedure to DROP a job, DROP any arguments that belong to the job, and eliminate any future job executions. The signature of the procedure is:

DROP\_JOB(  
 *job\_name* IN VARCHAR2,  
 *force* IN BOOLEAN DEFAULT FALSE,  
 *defer* IN BOOLEAN DEFAULT FALSE,  
 *commit\_semantics* IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

**Parameters**

*job\_name*

*job\_name* specifies the name of the job that is being dropped.

*force*

*force* is accepted for compatibility, and ignored.

*defer*

*defer* is accepted for compatibility, and ignored.

*commit\_semantics*

*commit\_semantics* instructs the server how to handle an error encountered while dropping a program or job. By default, *commit\_semantics* is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

**Example**

The following call to DROP\_JOB drops a job named update\_log:

DBMS\_SCHEDULER.DROP\_JOB('update\_log');

**7.12.8****DROP\_PROGRAM**

The DROP\_PROGRAM procedure

The signature of the DROP\_PROGRAM procedure is:

DROP\_PROGRAM(  
 *program\_name* IN VARCHAR2,  
 *force* IN BOOLEAN DEFAULT FALSE)

**Parameters**

*program\_name*

*program\_name* specifies the name of the program that is being dropped.

*force*

*force* is a BOOLEAN value that instructs the server how to handle programs with dependent jobs.

Specify FALSE to instruct the server to return an error if the program is referenced by a job.

Specify TRUE to instruct the server to disable any jobs that reference the program before dropping the program.

The default value is FALSE.

**Example**

The following call to DROP\_PROGRAM drops a job named update\_emp:

DBMS\_SCHEDULER.DROP\_PROGRAM('update\_emp');

**7.12.9****DROP\_PROGRAM\_ARGUMENT**

Use the DROP\_PROGRAM\_ARGUMENT procedure to drop a program argument. The DROP\_PROGRAM\_ARGUMENT procedure comes in two forms; the first form uses an argument position to specify which argument to drop:

DROP\_PROGRAM\_ARGUMENT(  
 *program*\_*name* IN VARCHAR2,  
 *argument\_position* IN PLS\_INTEGER)

The second form takes the argument name:

DROP\_PROGRAM\_ARGUMENT(  
 *program*\_*name* IN VARCHAR2,  
 *argument\_name* IN VARCHAR2)

**Parameters**

*program\_name*

*program\_name* specifies the name of the program that is being modified.

*argument\_position*

*argument\_position* specifies the position of the argument that is being dropped.

*argument\_name*

*argument\_name* specifies the name of the argument that is being dropped.

**Examples**

The following call to DROP\_PROGRAM\_ARGUMENT drops the first argument in the update\_emp program:

DBMS\_SCHEDULER.DROP\_PROGRAM\_ARGUMENT('update\_emp', 1);

The following call to DROP\_PROGRAM\_ARGUMENT drops an argument named emp\_name:

DBMS\_SCHEDULER.DROP\_PROGRAM\_ARGUMENT(update\_emp', 'emp\_name');

**7.12.10****DROP\_SCHEDULE**

Use the DROP\_SCHEDULE procedure to drop a schedule. The signature is:

DROP\_SCHEDULE(  
 *schedule\_name* IN VARCHAR2,  
 *force* IN BOOLEAN DEFAULT FALSE)

**Parameters**

*schedule\_name*

*schedule\_name* specifies the name of the schedule that is being dropped.

*force*

*force* specifies the behavior of the server if the specified schedule is referenced by any job:

|  |  |
| --- | --- |
| • | Specify FALSE to instruct the server to return an error if the specified schedule is referenced by a job. This is the default behavior. |
| • | Specify TRUE to instruct the server to disable to any jobs that use the specified schedule before dropping the schedule. Any running jobs will be allowed to complete before the schedule is dropped. |

**Example**

The following call to DROP\_SCHEDULE drops a schedule named weeknights\_at\_5:

DBMS\_SCHEDULER.DROP\_SCHEDULE('weeknights\_at\_5', TRUE);

The server will disable any jobs that use the schedule before dropping the schedule.

**7.12.11****ENABLE**

Use the ENABLE procedure to enable a disabled program or job.

The signature of the ENABLE procedure is:

ENABLE(  
 *name* IN VARCHAR2,  
 *commit\_semantics* IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

**Parameters**

*name*

*name* specifies the name of the program or job that is being enabled.

*commit\_semantics*

*commit\_semantics* instructs the server how to handle an error encountered while enabling a program or job. By default, *commit\_semantics* is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

**Example**

The following call to DBMS\_SCHEDULER.ENABLE enables the update\_emp program:

DBMS\_SCHEDULER.ENABLE('update\_emp');

**7.12.12****EVALUATE\_CALENDAR\_STRING**

Use the EVALUATE\_CALENDAR\_STRING procedure to evaluate the *repeat\_interval* value specified when creating a schedule with the CREATE\_SCHEDULE procedure. The EVALUATE\_CALENDAR\_STRING procedure will return the date and time that a specified schedule will execute without actually scheduling the job.

The signature of the EVALUATE\_CALENDAR\_STRING procedure is:

EVALUATE\_CALENDAR\_STRING(  
 *calendar*\_*string* IN VARCHAR2,  
 *start\_date* IN TIMESTAMP WITH TIME ZONE,  
 *return\_date\_after* IN TIMESTAMP WITH TIME ZONE,  
 *next\_run\_date* OUT TIMESTAMP WITH TIME ZONE)

**Parameters**

*calendar*\_*string*

*calendar\_string* is the calendar string that describes a *repeat\_interval* (see [Section 7.12.1](https://www.enterprisedb.com/edb-docs/d/edb-postgres-advanced-server/user-guides/database-compatibility-for-oracle-developers-guide/9.4/Database_Compatibility_for_Oracle_Developers_Guide.1.188.html#pID0E0AMJ0HA)) that is being evaluated.

*start\_date* IN TIMESTAMP WITH TIME ZONE

*start\_date* is the date and time after which the *repeat\_interval* will become valid.

*return\_date\_after*

Use the *return\_date\_after parameter* to specify the date and time that EVALUATE\_CALENDAR\_STRING should use as a starting date when evaluating the *repeat\_interval*.

For example, if you specify a *return\_date\_after* value of 01-APR-13 09.00.00.000000, EVALUATE\_CALENDAR\_STRING will return the date and time of the first iteration of the schedule after April 1st, 2013.

*next\_run\_date* OUT TIMESTAMP WITH TIME ZONE

*next\_run\_date* is an OUT parameter that will contain the first occurrence of the schedule after the date specified by the *return\_date\_after* parameter.

**Example**

The following example evaluates a calendar string and returns the first date and time that the schedule will be executed after June 15, 2013:

DECLARE

result TIMESTAMP;

BEGIN

DBMS\_SCHEDULER.EVALUATE\_CALENDAR\_STRING

(

'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',

'15-JUN-2013', NULL, result

);

DBMS\_OUTPUT.PUT\_LINE('next\_run\_date: ' || result);

END;

/

next\_run\_date: 17-JUN-13 05.00.00.000000 PM

June 15, 2013 is a Saturday; the schedule will not execute until Monday, June 17, 2013 at 5:00 pm.

**7.12.13****RUN\_JOB**

Use the RUN\_JOB procedure to execute a job immediately. The signature of the RUN\_JOB procedure is:

RUN\_JOB(  
 *job\_name* IN VARCHAR2,  
 *use\_current\_session* IN BOOLEAN DEFAULT TRUE

**Parameters**

*job\_name*

*job\_name* specifies the name of the job that will execute.

*use\_current\_session*

By default, the job will execute in the current session. If specified, *use\_current\_session* must be set to TRUE ; if *use\_current\_session* is set to FALSE, Advanced Server will return an error.

**Example**

The following call to RUN\_JOB executes a job named update\_log:

DBMS\_SCHEDULER.RUN\_JOB('update\_log', TRUE);

Passing a value of TRUE as the second argument instructs the server to invoke the job in the current session.

**7.12.14****SET\_JOB\_ARGUMENT\_VALUE**

Use the SET\_JOB\_ARGUMENT\_VALUE procedure to specify a value for an argument. The SET\_JOB\_ARGUMENT\_VALUE procedure comes in two forms; the first form specifies which argument should be modified by position:

SET\_JOB\_ARGUMENT\_VALUE(  
 *job\_name* IN VARCHAR2,  
 *argument\_position* IN PLS\_INTEGER,  
 *argument\_value* IN VARCHAR2)

The second form uses an argument name to specify which argument to modify:

SET\_JOB\_ARGUMENT\_VALUE(  
 *job\_name* IN VARCHAR2,  
 *argument\_name* IN VARCHAR2,  
 *argument\_value* IN VARCHAR2)

Argument values set by the SET\_JOB\_ARGUMENT\_VALUE procedure override any values set by default.

**Parameters**

*job\_name*

*job\_name* specifies the name of the job to which the modified argument belongs.

*argument\_position*

Use *argument\_position* to specify the argument position for which the value will be set.

*argument\_name*

Use *argument\_name* to specify the argument by name for which the value will be set.

*argument\_value*

*argument\_value* specifies the new value of the argument.

**Examples**

The following example assigns a value of 30 to the first argument in the update\_emp job:

DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE('update\_emp', 1, '30');

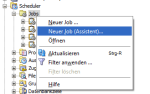
The following example sets the emp\_name argument to SMITH:

DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE('update\_emp', 'emp\_name', 'SMITH');

==================

scheduled job on procedure with DATE parameters

[JULY 3, 2015](https://svenweller.wordpress.com/2015/07/03/scheduled-job-on-procedure-with-date-parameters/) [SVENWELLER](https://svenweller.wordpress.com/author/svenweller/)[2 COMMENTS](https://svenweller.wordpress.com/2015/07/03/scheduled-job-on-procedure-with-date-parameters/#comments)

To set up a scheduled job in an oracle database is really simple. Oracle SQL Developer helps a lot to do so. It has a nice configuration wizard that creates all the needed function calls to DBMS\_SCHEDULER for you.  
[](https://svenweller.files.wordpress.com/2015/07/scheduler_date_1.png)

But some certain small issues can still be a hassle. For example if you want to run a procedure with parameters. Number and string parameters can be passed using [DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE](https://docs.oracle.com/database/121/ARPLS/d_sched.htm#i1011390). If you use sql developer to create the job for you, then it will use this function for all types of parameters.

That means you can’t easily pass sysdate. Because that is simply passed as a string.

This is the code that SQL developer generates.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | SYS.DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE(  job\_name => '"MYSCHEMA"."myScheduledJob"',  argument\_position => 1,  argument\_value => '10');  SYS.DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE(  job\_name => '"MYSCHEMA"."myScheduledJob"',  argument\_position => 2,  argument\_value => 'sysdate');  SYS.DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE(  job\_name => '"MYSCHEMA"."myScheduledJob"',  argument\_position => 3,  argument\_value => 'false'); |

It is less known that you can also pass other types of parameters using [DBMS\_SCHEDULER.SET\_JOB\_ANYDATA\_VALUE](https://docs.oracle.com/database/121/ARPLS/d_sched.htm#i1000820). It accepts an anydata data type. [Anydata](http://docs.oracle.com/database/121/ARPLS/t_anydat.htm" \l "ARPLS077" \t "_blank) is like a placeholder for any scalar data type. You put a certain value or expression in and it “stores” the value and the data type.

I always change the job\_name parameters from SQL Developer and replace it with a variable.  
Here is an example of a real job that I set up using a date parameter value (value = “in 10 minutes”). always change the job\_name parameters from SQL Developer and replace it with a variable.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | -- create job  declare  v\_jobname varchar2(30) := 'myScheduledJob';  v\_date\_format varchar2(100);  BEGIN  select value  into v\_date\_format  from v$nls\_parameters  where parameter = 'NLS\_DATE\_FORMAT';    SYS.DBMS\_SCHEDULER.CREATE\_JOB (  job\_name => v\_jobname,  job\_type => 'STORED\_PROCEDURE',  job\_action => 'MYSCHEMA.MYPACKAGE.RUN\_BATCH',  number\_of\_arguments => 2,  start\_date => TO\_TIMESTAMP\_TZ('2015-06-03 19:05:00 Europe/Berlin','YYYY-MM-DD HH24.MI.SS TZR'),  --start\_date => systimestamp + interval '1' minute,  repeat\_interval => NULL,  --end\_date => TO\_TIMESTAMP\_TZ('2015-07-29 05:30:00 Europe/Berlin','YYYY-MM-DD HH24.MI.SS TZR'),  job\_class => '"SYS"."DEFAULT\_JOB\_CLASS"',  enabled => FALSE,  auto\_drop => FALSE,  comments => 'Do run something longrunning');    SYS.DBMS\_SCHEDULER.SET\_JOB\_ARGUMENT\_VALUE(  job\_name => v\_jobname,  argument\_position => 1,  argument\_value => 10);    SYS.DBMS\_SCHEDULER.SET\_JOB\_ANYDATA\_VALUE(  job\_name => v\_jobname,  argument\_position => 2,  argument\_value => sys.anydata.convertDate(sysdate + interval '10' min)  );    SYS.DBMS\_SCHEDULER.SET\_ATTRIBUTE(  name => v\_jobname,  attribute => 'logging\_level', value => DBMS\_SCHEDULER.LOGGING\_RUNS);    SYS.DBMS\_SCHEDULER.SET\_ATTRIBUTE(  name =>v\_jobname,  attribute => 'max\_run\_duration', value => INTERVAL '1' DAY);    SYS.DBMS\_SCHEDULER.SET\_ATTRIBUTE(  name => v\_jobname,  attribute => 'schedule\_limit', value => INTERVAL '1' DAY);    SYS.DBMS\_SCHEDULER.enable(  name => v\_jobname);  commit;  END;  / |

The expression “anydata.convertDate(sysdate + interval ’10’ min)” was used to set the parameter value.

Feel free to use that as a template for your own scheduled jobs.

-=========================

# Running Procedures Asynchronously with Oracle Job Scheduler

*JANUARY 13, 2019*

Consider a PL/SQL stored procedure that handles a heavy transaction. The procedure is extremely slow – when executed from a UI, the application hangs for minutes. On analysis it is found that the procedure is performing a complex series of steps, a portion of which are non-critical and need not hold up the entire transaction. In other words, it would be acceptable if:

* some of the steps are run asynchronously while a slimmer main transaction completes
* failures (if any) in the asynchronous steps do not cause a failure in the main transaction

Oracle PL/SQL helps us achieve these objectives with asynchronous processing using Oracle job scheduler DBMS\_SCHEDULER. Here’s a demo to show you how.

### Use Case for Demo: Movie Ticketing Application

Let’s say this heavy and slow procedure is one that creates a booking in a movie ticketing application. In its current form, it does all of the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | create\_booking (booking\_id)  {    allocate\_seats;    capture\_customer\_details;    receive\_payment;    notify\_customer;    upsell\_food;    update\_central\_crm;  } |

# The problem: The last three “non-critical” steps are slowing down the main flow. While it is desirable to have these steps work close to real-time, some lag with these steps is acceptable if it makes the main flow of create\_booking faster.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | -- BEFORE Slow create\_booking  -- All processing synchronous  create\_booking (booking\_id)  {      -- Critical parts of booking: main flow, any failure    -- here must fail the entire booking    allocate\_seats;    capture\_customer\_details;    receive\_payment;      -- Non-critical parts of booking: desirable to have this    -- as close as possible to real-time but not at the cost    -- of slowing down the main flow    notify\_customer;    upsell\_food;    update\_central\_crm;   } |

# The solution: Move the last three “non-critical” steps to another procedure and invoke that procedure asynchronously. The final state should look like this algorithmically:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | -- AFTER Fast create\_booking  -- Non-critical processing asynchronous  create\_booking (booking\_id)  {      -- Critical parts of booking: main flow, any failure    -- here must fail the entire booking    allocate\_seats;    capture\_customer\_details;    receive\_payment;      -- Non-critical parts of booking: wrapped in    -- a separate procedure called asynchronously    [async] post\_booking\_flow(booking\_id);   }    -- Async steps factored out into separate procedure  post\_booking\_flow (booking\_id)  {    notify\_customer;    upsell\_food;    update\_central\_crm;  } |

# Oracle Job Scheduler to Run Procedure Asynchronously: Summary

# To run a piece of code asynchronously via Oracle Job Scheduler, a summary of the steps needed:

# Create procedure to be run asynchronously

# Call the procedure via DBMS\_SCHEDULER.[CREATE\_JOB](https://docs.oracle.com/en/database/oracle/oracle-database/12.2/arpls/DBMS_SCHEDULER.html#GUID-7E744D62-13F6-40E9-91F0-1569E6C38BBC) in the main flow for a single immediate run (i.e. AUTO\_DROP = TRUE, the default)

# Oracle Job Scheduler to Run Job Asynchronously: Stepwise Scripts

# 1. Create procedure to be run asynchronously

# This procedure is a wrapper to the portion of code in the main flow which is to be run asynchronously.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | -- Procedure to be run asynchronously  create or replace procedure  post\_booking\_flow (booking\_id in varchar2)  as  begin    dbms\_output.put\_line('START post\_booking\_flow');    notify\_customer;    upsell\_food;    update\_central\_crm;    dbms\_output.put\_line('END post\_booking\_flow');  end;  / |

# 2. Call the procedure via DBMS\_SCHEDULER.CREATE\_JOB for a single immediate run

# The parameter values for CREATE\_JOB to support single immediate run:

# *job\_name:*A unique name, say ‘post\_booking\_flow\_job’ concatenated with booking\_id to prevent concurrency conflict

# *job\_type:* PLSQL\_BLOCK

# *job\_action:* PL/SQL block invoking procedure created in step 1

# *enabled:* TRUE (default is FALSE)

# *auto\_drop:* TRUE (default is TRUE)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | -- Non-critical processing asynchronous  -- with DBMS\_SCHEDULER.CREATE\_JOB  create or replace procedure create\_booking  (booking\_id in varchar2)  as  begin    dbms\_output.put\_line('START create\_booking');    -- Critical parts of booking: main flow, any failure    -- here must fail the entire booking    allocate\_seats;    capture\_customer\_details;    receive\_payment;      -- Non-critical parts of booking: wrapped in    -- a separate procedure called asynchronously    dbms\_output.put\_line('Before post\_booking\_flow\_job');    dbms\_scheduler.create\_job (    job\_name   =>  'post\_booking\_flow\_job'||booking\_id,    job\_type   => 'PLSQL\_BLOCK',    job\_action =>      'BEGIN         post\_booking\_flow('''||booking\_id||''');       END;',    enabled   =>  TRUE,    auto\_drop =>  TRUE,    comments  =>  'Non-critical post-booking steps');      dbms\_output.put\_line('After post\_booking\_flow\_job');    dbms\_output.put\_line('END create\_booking');  end;  / |

# Oracle Job Scheduler to Run Job Asynchronously: Scripts When Run

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | SQL> -- Procedure to be run asynchronously  SQL> create or replace procedure    2  post\_booking\_flow (booking\_id in varchar2)    3  as    4  begin    5    dbms\_output.put\_line('START post\_booking\_flow');    6    notify\_customer;    7    upsell\_food;    8    update\_central\_crm;    9    dbms\_output.put\_line('END post\_booking\_flow');   10  end;   11  /    Procedure created. |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | SQL> -- Non-critical processing asynchronous  SQL> -- with DBMS\_SCHEDULER.CREATE\_JOB  SQL> create or replace procedure create\_booking    2  (booking\_id in varchar2)    3  as    4  begin    5    dbms\_output.put\_line('START create\_booking');    6    -- Critical parts of booking: main flow, any failure    7    -- here must fail the entire booking    8    allocate\_seats;    9    capture\_customer\_details;   10    receive\_payment;   11   12    -- Non-critical parts of booking: wrapped in   13    -- a separate procedure called asynchronously   14    dbms\_output.put\_line('Before post\_booking\_flow\_job');   15    dbms\_scheduler.create\_job (   16    job\_name   =>  'post\_booking\_flow\_job'||booking\_id,   17    job\_type   => 'PLSQL\_BLOCK',   18    job\_action =>   19      'BEGIN   20         post\_booking\_flow('''||booking\_id||''');   21       END;',   22    enabled   =>  TRUE,   23    auto\_drop =>  TRUE,   24    comments  =>  'Non-critical post-booking steps');   25   26    dbms\_output.put\_line('After post\_booking\_flow\_job');   27    dbms\_output.put\_line('END create\_booking');   28  end;   29  /    Procedure created. | |

# Test the asynchronous PL/SQL job

# Pass the argument booking\_id and test create\_booking, now branching off into an asynchronous call to post\_booking\_flow\_job||booking\_id.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | SQL>-- Running the procedure with async call  SQL> exec create\_booking('A001');  START create\_booking  Before post\_booking\_flow\_job  After post\_booking\_flow\_job  END create\_booking    PL/SQL procedure successfully completed.    SQL> -- Running the procedure with async call  SQL> exec create\_booking('A002');  START create\_booking  Before post\_booking\_flow\_job  After post\_booking\_flow\_job  END create\_booking    PL/SQL procedure successfully completed. |

# Good news: the asynchronous portion of create\_booking code no longer causes a lag in the booking process.

# Post Async Job Run: Verify Scheduler Logs

# ALL\_SCHEDULER\_JOB\_LOG displays log for the Scheduler jobs accessible to the current user.

# ALL\_SCHEDULER\_JOB\_RUN\_DETAILS displays run details for the Scheduler jobs accessible to the current user.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | SQL> -- Job logs generated in the last hour  SQL> select job\_name, status    2  from all\_scheduler\_job\_log    3  where job\_name like 'POST\_BOOKING\_FLOW\_JOB%'    4  and log\_date > sysdate - 1/24    5  order by log\_date desc;    JOB\_NAME                  STATUS  ------------------------- ----------  POST\_BOOKING\_FLOW\_JOBA002 SUCCEEDED  POST\_BOOKING\_FLOW\_JOBA001 SUCCEEDED    SQL> -- Job run details generated in the last hour  SQL> select job\_name, status    2  from all\_scheduler\_job\_run\_details    3  where job\_name like 'POST\_BOOKING\_FLOW\_JOB%'    4  and log\_date > sysdate - 1/24    5  order by log\_date desc;    JOB\_NAME                  STATUS  ------------------------- ----------  POST\_BOOKING\_FLOW\_JOBA002 SUCCEEDED  POST\_BOOKING\_FLOW\_JOBA001 SUCCEEDED |

# Scheduler Privileges to Create Job Asynchronously

# In addition to EXECUTE privilege on the package DBMS\_SCHEDULER, one must have:

# CREATE JOB privilege to create a job in one’s own [schema](https://www.oratable.com/oracle-user-schema-difference/), and

# CREATE ANY JOB privilege to create a job in any schema except SYS

# To invoke a PL/SQL program when creating a job, the owner of the job must be the owner of the program or have EXECUTE privilege on the program.

# For Further Reading

# Running procedures asynchronously is just one use case for DBMS\_SCHEDULER – look up [Oracle 12c documentation](https://docs.oracle.com/en/database/oracle/oracle-database/12.2/admin/scheduling-jobs-with-oracle-scheduler.html) for the range of its features and utilities.

# Also see: [Get job name inside DBMS\_SCHEDULER executed procedure](http://stevenfeuersteinonplsql.blogspot.com/2014/08/plsql-challenge-community-to-rescue-get.html)

# =======================

Do you know a way to get the DBMS\_SCHEDULER Job Name from within the code that is being executed by the job?

I could have told him to visit the OTN SQL and PL/SQL Forum and post his question there, but I thought that instead I would ask the players at the PL/SQL Challenge if they had any ideas. So I posted this message in the Recent News section:

**Solution from Niels Hecker**

-- the user needs the following privileges directly assigned:

-- EXECUTE on package DBMS\_Lock

-- SELECT ANY DICTIONARY

-- CREATE JOB

------------------------------------------------------------------

-- create logging-table with associated log-procedure

CREATE TABLE tbl\_LogMsg (

ID INTEGER,

Stamp TIMESTAMP(3),

Msg VARCHAR2(4000)

);

CREATE SEQUENCE seq\_LogMsg#ID

START WITH 0 INCREMENT BY 1

MINVALUE 0 MAXVALUE 4294967295

ORDER NOCACHE NOCYCLE;

CREATE OR REPLACE

PROCEDURE LogMsg (pMsg IN VARCHAR2)

IS PRAGMA AUTONOMOUS\_TRANSACTION;

tmp VARCHAR2(4000);

BEGIN

tmp := RTrim( SubStr( pMsg, 1, 4000));

INSERT INTO tbl\_LogMsg (ID, Stamp, Msg)

VALUES (seq\_LogMsg#ID.NEXTVAL, SYSTIMESTAMP, tmp);

COMMIT WORK;

EXCEPTION

WHEN OTHERS THEN ROLLBACK WORK;

END LogMsg;

/

------------------------------------------------------------------

-- create a job to execute the procedure (created in the next step)

BEGIN

DBMS\_Scheduler.Create\_Job(

Job\_Name => '"This is the Job to look after"',

Job\_Type => 'PLSQL\_BLOCK',

Job\_Action => 'ExecutedAsJob();',

Start\_Date => NULL,

Repeat\_Interval => NULL,

Enabled => False,

Auto\_Drop => False,

Comments => 'Just a test to see if you can find out the job-name'

);

END;

/

------------------------------------------------------------------

-- create the full procedure

CREATE OR REPLACE

PROCEDURE ExecutedAsJob

IS

iJobID INTEGER;

vcOwner VARCHAR2(30);

vcName VARCHAR2(30);

FUNCTION GetJobObjectID$ RETURN INTEGER

IS

Result INTEGER;

iSID INTEGER;

iInstance INTEGER;

BEGIN

-- retrieve the actual session-id and the instance-number

Result := Sys\_Context( 'UserEnv', 'BG\_Job\_ID');

IF (Result IS NULL) THEN

iSID := Sys\_Context( 'UserEnv', 'SID');

iInstance := Sys\_Context( 'UserEnv', 'Instance');

-- retrieve the id of the actual job which is in fact

-- the object-id of the scheduler job

-- (assertion: there is only one running job at a time

-- for the a specific session)

SELECT srj.Job\_ID

INTO Result

FROM gv$Scheduler\_Running\_Jobs srj

WHERE (srj.Inst\_ID = iInstance)

AND (srj.Session\_ID = iSID);

LogMsg( 'GetJobObjectID$() - SID: ' || iSID || ', Instance: '

|| iInstance);

END IF; -- (Result IS ...

RETURN (Result);

EXCEPTION

WHEN OTHERS THEN

LogMsg( SQLERRM);

LogMSg( DBMS\_Utility.Format\_Error\_BackTrace());

RETURN (NULL);

END GetJobObjectID$; -- local to ExecutedAsJob

BEGIN -- of ExcecutedAsJob

LogMsg( 'Procedure/Job started');

DBMS\_Lock.Sleep( 1.0);

-- get the job-/object-id

iJobID := GetJobObjectID$();

IF (iJobID IS NOT NULL) THEN

LogMsg( 'Found this running job - ID: ' || iJobID);

DBMS\_Lock.Sleep( 1.5);

ELSE

LogMsg( 'Ooops - no running job found and goodbye ...');

RETURN;

END IF;

-- get the owner and name of the job

SELECT o.Owner, o.Object\_Name

INTO vcOwner, vcName

FROM DBA\_Objects o

WHERE (o.Object\_ID = iJobID);

LogMsg( 'Job-Object: "' || vcOwner || '"."' || vcName || '"');

DBMS\_Lock.Sleep( 2.5);

LogMsg( 'Procedure/Job ended');

EXCEPTION

WHEN OTHERS THEN LogMsg( SQLERRM);

END ExecutedAsJob;

/

------------------------------------------------------------------

-- code to run the job and query to see the results

BEGIN

DELETE FROM tbl\_LogMsg;

COMMIT WORK;

DBMS\_Scheduler.Enable( '"This is the Job to look after"');

END;

/

SELECT Sys\_Context( 'UserEnv', 'SID') AS "SID", t.\*

FROM tbl\_LogMsg t ORDER BY t.ID;

-- after 5 seconds the query should give a result like:

/\*

SID ID STAMP MSG

---- --- ------------------------ ----------------------------------------------------

66 0 2014-08-07 11:26:06,930 Procedure/Job started

66 1 2014-08-07 11:26:07,932 Found this running job - ID: 266922

66 2 2014-08-07 11:26:09,434 Job-Object: "TEST2"."This is the Job to look after"

66 3 2014-08-07 11:26:11,935 Procedure/Job ended

\*/

------------------------------------------------------------------

-- clean up the database

/\*

exec DBMS\_Scheduler.Drop\_Job( '"This is the Job to look after"');

DROP PROCEDURE ExecutedAsJob;

DROP PROCEDURE LogMsg;

DROP SEQUENCE seq\_LogMsg#ID;

DROP TABLE tbl\_LogMsg PURGE;

\*/

The fellow in need reported that this did solve his problem. Thanks, Niel!

Of course, the Oracle Dev Gym community is full of helpful and expert folks, so I did receive other ideas as well. I offer them below, with the caveat that I have not tested them myself. Thanks to everyone for their assistance!

**From Chris Saxon**

Hi Steven,

I saw your post about the getting the name of a job from within it on PLCH. This approach does that.

The view USER\_SCHEDULER\_RUNNING\_JOBS show jobs that are active. Querying this, filtering on the current session id will return the current job (if you're within one). The query is:

**SELECT job\_name FROM user\_scheduler\_running\_jobs**

**where  session\_id = sys\_context('USERENV', 'SID');**

A full script to show this is below

--

Thanks,

Chris

[www.sqlfail.com](http://www.sqlfail.com/)

**create table job\_name ( name varchar2(100) )**

**/**

**create or replace procedure store\_job as**

**begin**

**insert into job\_name ( name )**

**SELECT job\_name FROM user\_scheduler\_running\_jobs**

**where  session\_id = sys\_context('USERENV', 'SID');**

**commit;**

**end;**

**/**

**BEGIN**

**DBMS\_SCHEDULER.CREATE\_JOB (**

**job\_name => '"CHRIS"."TEST\_JOB"',**

**job\_type => 'STORED\_PROCEDURE',**

**job\_action => 'CHRIS.STORE\_JOB',**

**number\_of\_arguments => 0,**

**start\_date => NULL,**

**repeat\_interval => NULL,**

**end\_date => NULL,**

**enabled => FALSE,**

**auto\_drop => FALSE,**

**comments => '');**

**DBMS\_SCHEDULER.enable(**

**name => '"CHRIS"."TEST\_JOB"');**

**SYS.dbms\_scheduler.run\_job ('TEST\_JOB');**

**END;**

**/**

**SELECT \* FROM job\_name**

**/**

**From Cristi Boboc**

I do not have much experience either with this package but I think the name of the job can be obtained by the following algorithm:

1. I get the session under which the process execute,

2. From the active running Jobs I get the one which runs in the same session.

A pseudo-code (I do not have an environment to test - therefore kindly please excuse my mistakes) could look like:  
  
**SELECT owner, job\_name, running\_instance, session\_id, j.\***

**FROM all\_scheduler\_running\_jobs j  
WHERE session\_id = sys\_context('USERENV','SID')**

or, if the "old way of scheduling jobs":  
  
**SELECT job, instance, sid, j.\* FROM dba\_jobs\_running j  
WHERE session\_id = sys\_context('USERENV','SID')**

**From Zoltan Fulop**

You raised a question on PL/SQL Challenge regarding DBMS\_SCHEDULER. Since I worked a lot with that package let me share my experience about how to get the job name within the code that is being executed by the job. You can use the dictionary view called user\_scheduler\_running\_jobs which lists the currently running jobs or you can simply get the v$session.action attribute by the sys\_context('USERENV', 'ACTION') if you're running that job in a background process. Here you can find an example:  
  
**CREATE TABLE plch\_log (job\_name VARCHAR2(100));  
  
CREATE OR REPLACE PROCEDURE plch\_proc  
IS  
  l\_job\_name user\_scheduler\_running\_jobs.job\_name%TYPE;  
BEGIN  
  BEGIN  
    SELECT job\_name  
      INTO l\_job\_name  
      FROM user\_scheduler\_running\_jobs  
     WHERE running\_instance = SYS\_CONTEXT ('USERENV', 'INSTANCE')  
       AND session\_id = SYS\_CONTEXT ('USERENV', 'SID');  
  EXCEPTION  
    WHEN no\_data\_found THEN  
      l\_job\_name := SYS\_CONTEXT ('USERENV', 'ACTION');  
  END;  
  
  INSERT INTO plch\_log VALUES (l\_job\_name);  
    
  COMMIT;  
END;  
/  
  
DECLARE  
  l\_job\_name VARCHAR2(100) := dbms\_scheduler.generate\_job\_name('PLCH\_');  
BEGIN  
  DBMS\_SCHEDULER.create\_job(  
    job\_name            => l\_job\_name  
   ,job\_type            => 'STORED\_PROCEDURE'  
   ,job\_action          => 'PLCH\_PROC'  
   ,enabled             => TRUE  
   ,auto\_drop           => TRUE);  
     
   DBMS\_SCHEDULER.run\_job(  
    job\_name            => l\_job\_name  
   ,use\_current\_session => FALSE);  
END;  
/  
  
BEGIN  
  DBMS\_LOCK.sleep(5);  
END;  
/  
  
SELECT \* FROM plch\_log;  
  
DROP PROCEDURE plch\_proc;  
  
DROP TABLE plch\_log;**

**From Iudith Mentzel**

I have zero experience with DBMS\_SCHEDULER, but, on a quick glance, maybe the following could help (I did not try it, it is just a "dry" idea)::

**SELECT JOB\_NAME  
  FROM  USER\_SCHEDULER\_RUNNING\_JOBS  
 WHERE SESSION\_ID       = SYS\_CONTEXT('USERENV','SID')  
   AND   RUNNING\_INSTANCE = SYS\_CONTEXT('USERENV','INSTANCE')  
/**

Another way would be to use the DBMS\_SCHEDULER.DEFINE\_METADATA\_ARGUMENT, which can pass the JOB\_NAME (and other job metadata ) to the program executed by the job, but, as far as I understand, the program should be prepared/defined to accept that argument, so it is maybe less generic.

**\*About that reference to Oracle Database Developer**

I used to talk about PL/SQL developers and APEX developers and SQL developer and so on, but I have recently come to realize that very, very few Oracle technologists can be “pigeon-holed” that way. Sure, *Steven* knows and uses only PL/SQL (and SQL), but just about everyone else on the planet relies on a whole smorgasbord of tools to build applications against Oracle Database. So I’m going to start referring to all of us simply as Oracle Database Developers.

# =======================

FND\_FILE.PUT\_LINE, will print the string in your output or LOG file, when you run a concurrent program.

FND\_FILE.PUT\_LINE(FND\_FILE.output, 'message'); -- This will print in Concurrent program output

FND\_FILE.PUT\_LINE(FND\_FILE.log, 'message'); -- This will print in Concurrent program log

UTL\_FILE.PUT\_LINE is going to print the string in a writable File, which you need to open in your pl-sql.

Example of using UTL\_FILE

v\_chr\_out\_file UTL\_FILE.file\_type;

v\_chr\_out\_file :=

UTL\_FILE.fopen (<directory\_path>,

<file\_name>,

'W',

32767);

UTL\_FILE.put\_line (v\_chr\_out\_file, 'this will get written in file');

UTL\_FILE.fclose (v\_chr\_out\_file);

# =================

FND\_FILE is built-in package provided in Oracle Apps/E-Business Suite to write log and output file in PL/SQL concurrent program. This is actually a wrapper on UTL\_FILE to write files on the Unix server. You can open the log and output file using navigation View -> Requests -> Find. Select the respective request on the summary page and click View Log or View Output for check respective log or output file.

* **Log File** – These are log files. Dump error message, debug message in this file.
* **Output File** – This file you can have some output of processing of the concurrent program.

In this article, I will share how to use fnd\_file to create log and output file.

To explain this API, we will use [PL/SQL concurrent Program in Oracle Apps](https://www.atechtown.com/define-concurrent-program-oracle-applications/). It is simple PL/SQL procedure which just write dummy message in log and output file.

## FND\_FILE

This package contains all the procedure and functions to write log and output file. Here I am just going to explain fnd\_file.put\_line procedure with example. You can refer below article from Oracle Apps Developer Guide to check all procedure and function available in fnd\_file package. This article is very details and does contain working exampe.

Read:- [PL/SQL APIs for Concurrent Processing](https://docs.oracle.com/cd/E18727_01/doc.121/e12897/T302934T458258.htm#I_fndfile)

Below are global variables defined in package specification to denote log or output file.

**Global Variables**

**FND\_FILE.LOG** – 1 – This is constant variable to denote log file

**FND\_FILE.OUTPUT** – 2  – This is constant variable to denote log file

### FND\_FILE.PUT\_LINE

**T**his procedure writes a line of text to a file including new line.

#### Syntax

procedure FND\_FILE.PUT\_LINE

(which IN NUMBER,

buff IN VARCHAR2);

which - Log file or output file. Use either FND\_FILE.LOG or FND\_FILE.OUTPUT.

buff - Text to write.

**Example 1**

In this example, a concurrent program is defined on below procedure. This proceudure used fnd\_file.put\_line to log output and log. Depending on which parameter message are written. Even though it is wrapper on UTL\_FILE, you do not have to explicit open and close file. It is completely manager by Concurrent manager.

CREATE OR REPLACE PROCEDURE fnd\_file\_Demo(

o\_chr\_errbuf VARCHAR2 ,

o\_chr\_errcode VARCHAR2 )

IS

BEGIN

--- Creating log mesage----

fnd\_file.put\_line(fnd\_file.log,'This is log file ');

fnd\_file.put\_line(fnd\_file.log,'You can put error, debug message here ' || SQLERRM);

fnd\_file.put\_line(fnd\_file.log,'You can also use built in API ' || fnd\_message.get);

---Creating output message

fnd\_file.put\_line(fnd\_file.output,'This is output file');

fnd\_file.put\_line(fnd\_file.output,'You can use it to put summary information');

END;

Run the concurrent program and check log and output file generated.

**Sample log file**

+---------------------------------------------------------------------------+

Start of log messages from FND\_FILE

+---------------------------------------------------------------------------+

This is log file

You can put error, debug message here ORA-0000: normal, successful completion

You can also use built in API

+---------------------------------------------------------------------------+

End of log messages from FND\_FILE

+---------------------------------------------------------------------------+

**Sample output file**

This is output file

You can use it to put summary information

### Other Public Procedures/APIs

**FND\_FILE.PUT** – This procedure writes text to a file. There is no new line inserted. Multiple calls to FND\_FILE.PUT will produce concatenated text.

**FND\_FILE.NEW\_LINE – T**his procedure writes line terminators (new line characters) to a file. It should be used with fnd\_file.put if you want to write message on different line.

# ===========================

* **The directory does not exist**
* **The directory is full**
* **The directory does not have appropriate write permissions**
* **The UTL\_FILE\_DIR parameter is incorrect and is not pointing to the right directory**
* **The $APPLPTMP parameter does not match the UTL\_FILE\_DIR parameter**