[O'Reilly's CD bookshelfs](http://docstore.mik.ua/orelly/bookshelf.html) | [FreeBSD](http://docstore.mik.ua/manuals/freebsd.html) | [Linux](http://docstore.mik.ua/manuals/linux.html) | [Cisco](http://docstore.mik.ua/univercd/cc/td/doc/product/) | [PHP](http://php-manual.us/)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Oracle PL/SQL Programming Guide to Oracle 8i Features**   |  |  |  | | --- | --- | --- | | [Previous: 6.1 Triggers on Nested Table View Columns](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_01.htm) | [**Chapter 6 New Trigger Features in Oracle8i**](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_01.htm) | [Next: 6.3 Schema-Level Event Triggers](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_03.htm) |     **6.2 Database-Level Event Triggers**  Oracle8 *i* allows you to define triggers to respond to database-level events (also known as system-level events), including the following:   * Logon to and logoff from a schema * Startup and shutdown of the database * Response to a server error   Here is the syntax for these triggers:  CREATE [ OR REPLACE ] TRIGGER  *trigger\_name*  { BEFORE | AFTER }  { SERVERERROR |  LOGON | LOGOFF |  STARTUP | SHUTDOWN }  ON DATABASE  BEGIN    *pl/sql\_statements*  END;  To create a trigger at the database level, you must have the ADMINISTER DATABASE TRIGGER system privilege. You will also need privileges to access any of the external references in the trigger's PL/SQL code.  [Table 6.1](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_02.htm" \l "ch06-86706) lists the different database-level events on which you may define a trigger.   | Table 6.1: Database-Level Events for Trigger Definitions | | | --- | --- | | **Event** | **Description** | | SERVERERROR | Oracle fires the trigger whenever a server error message is logged. | | LOGON | Oracle fires the trigger after a client application logs on to the database successfully. | | LOGOFF | Oracle fires the trigger before a client application logs off the database. | | STARTUP | Oracle fires the trigger immediately after the database is opened. | | SHUTDOWN | Oracle fires the trigger just before the server starts a shutdown of an instance of the database. Note that a SHUTDOWN trigger will only fire if the database is shut down cleanly using SHUTDOWN or SHUTDOWN IMMEDIATE. If an instance shuts down abnormally (SHUTDOWN ABORT), this event will not be fired. |   Each database-level event has an associated with it a set of attributes. These attributes are actually functions owned by SYS that return the values of characteristics relevant to the event. [Table 6.2](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_02.htm" \l "ch06-47269) lists the current set of attributes.   | Table 6.2: Attributes for Database-Level Events | | | | --- | --- | --- | | **Name** | **Datatype** | **Description** | | SYSEVENT | VARCHAR2(30) | The database-level event firing the trigger; this value matches the name used in the syntax of the trigger. | | LOGIN\_USER | VARCHAR2(30) | The login username. | | INSTANCE\_NUM | NUMBER | The instance number. | | DATABASE\_NAME | VARCHAR2(50) | The name of the database. | | SERVER\_ERROR | NUMBER | This function returns the error at the *n* th position in the stack. You must specify a position (1 equals "top") when you call this function. For example:  SERVER\_ERROR(1) | | IS\_SERVERERROR | BOOLEAN | Returns TRUE if the specified error is on the current error stack; FALSE otherwise. |   [Table 6.3](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_02.htm" \l "ch06-45221) lists the restrictions and attributes for each database-level event.   | Table 6.3: Restrictions and Attributes for Database-Level Events | | | | --- | --- | --- | | **Event** | **Conditions/Restrictions** | **Attributes** | | SERVERERROR | You can specify a condition that will restrict the trigger to firing only when the specified exception is raised. Otherwise, it will fire for all errors. | SYSEVENT  LOGIN\_USER  INSTANCE\_NUM  DATABASE\_NAME  SERVER\_ERROR  IS\_SERVERERROR | | LOGON | You can specify a condition using either USERID( ) or USERNAME( ). | SYSEVENT  LOGIN\_USER  INSTANCE\_NUM  DATABASE\_NAME | | LOGOFF | You can specify a condition using either USERID( ) or USERNAME( ). | SYSEVENT  LOGIN\_USER  INSTANCE\_NUM  DATABASE\_NAME | | STARTUP | No database operations (DML and queries) are allowed. Instead, you might start up listener programs, pin code in memory, etc. | SYSEVENT  LOGIN\_USER  INSTANCE\_NUM  DATABASE\_NAME | | SHUTDOWN | No database operations (DML and queries) are allowed. Instead, you might shut down listener programs, run performance collection utilities that write logs to files, etc. | SYSEVENT  LOGIN\_USER  INSTANCE\_NUM  DATABASE\_NAME |   Keep in mind the following rules when working with these triggers:   * Whenever a database-level event trigger fires, Oracle opens an autonomous transaction, fires the trigger, and commits any DML in the trigger logic independently of the existing user transaction. * When defining LOGON, STARTUP, and SERVERERROR triggers, you can only specify the AFTER context. If you specify BEFORE, you will get this error:   ORA-30500: database open triggers and server error  triggers cannot have BEFORE type   * When defining LOGOFF and SHUTDOWN triggers, you can only specify the BEFORE context. If you specify AFTER, you will get this error:   ORA-30509: client logoff triggers cannot have AFTER type   * You cannot define AFTER STARTUP and BEFORE SHUTDOWN triggers for a schema; these apply only to DATABASE. * Calls to DBMS\_OUTPUT.PUT\_LINE do not generate any visible output in the current session. If you want to obtain a record of actions that occurred, you will need to write information to a database table, database pipe, or operating system file. You could also use DBMS\_AQ.ENQUEUE to place a message in a queue. * A SERVERERROR trigger will not fire for any of the following errors:   ORA-01403: no data found  ORA-01422: exact fetch returns more than requested number of rows  ORA-04030: out of process memory when trying to allocate  *nnn*  bytes  ORA-01034: ORACLE not available  ORA-01007: variable not in select list  **6.2.1 Examples of Database-Level Event Triggers**  There are many different ways you can use database-level event triggers to manage your database. The following examples offer some simple models from which you can build your own, more complex variations.  **6.2.1.1 Pinning packages on database startup**  A common requirement for a high-performance database is to pin one or more packages into shared memory. By pinning your code, it is exempted from the least recently used algorithm and will *never* be aged out of the shared pool area.  Before the existence of STARTUP triggers, the DBA would have to set up a script to run after the database was started. Now, I can create a STARTUP trigger like the following:  /\* Filename on companion disk: startup.trg \*/  CREATE OR REPLACE TRIGGER pin\_code\_on\_startup  AFTER STARTUP ON DATABASE  BEGIN  /\* Pin the default packages of PL/SQL to  improve runtime performance. \*/  DBMS\_SHARED\_POOL.KEEP (  'SYS.STANDARD', 'P');  DBMS\_SHARED\_POOL.KEEP (  'SYS.DBMS\_STANDARD', 'P');  END;  /  With this trigger in place, I guarantee that all of my large code elements (including cursors, types, and triggers) and even my sequences are cached in the shared pool until the instance goes down.  If you are going to be pinning objects, you should consider building an infrastructure table to store the names of elements you want pinned. Then instead of hard-coding your pin commands in a trigger (as just shown), you could run a procedure that reads through the table and pins each element found. This approach is explained in detail in *Chapter 12* of *Oracle Built-in Packages* .  **6.2.1.2 Tracking logins to the database**  Suppose that I want to keep track of logins to my database instance. To make this information available in a structured fashion, I am going to send my login information to a queue using the Oracle Advanced Queuing (AQ) facility.  These steps are illustrated in [Figure 6.2](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_02.htm" \l "ch06-12116) . You will find all of the AQ-related steps in the *aq.sql* script on the companion disk; I'll concentrate on the trigger-related components here.  **Figure 6.2: Logical flow of LOGON trigger**  Figure 6.2  Let's start in reverse. Here is the LOGON trigger:  /\* Filename on companion disk: aq.sql \*/  CREATE OR REPLACE TRIGGER publish\_logon  AFTER LOGON ON DATABASE  BEGIN  oraevent.put ('This is a logon');  END;  /  As you can see, there isn't much to it: every time someone logs in to the database, we'll put a message in the queue with a call to oraevent.put. So let's take a look at that procedure:  PROCEDURE oraevent.put (details\_in IN VARCHAR2)  IS  q\_opts DBMS\_AQ.ENQUEUE\_OPTIONS\_T;  msg\_props DBMS\_AQ.MESSAGE\_PROPERTIES\_T;  msg\_handle RAW(16);  event\_l Event\_t;  BEGIN  /\* Setting visibility to IMMEDIATE will  || force the queue to "commit" before the  || client transaction commits.  \*/  q\_opts.visibility := DBMS\_AQ.IMMEDIATE;  event\_l := Event\_t.make(details\_in);  DBMS\_AQ.ENQUEUE(queue\_name => 'aqadmin.loginQ',  enqueue\_options => q\_opts,  message\_properties => msg\_props,  payload => event\_l,  msgid => msg\_handle);  END;  This procedure calls DMBS\_AQ.ENQUEUE to place a message, or *payload* , in the loginQ queue. Each message is an object of type Event\_t, defined as follows:  CREATE TYPE Event\_t AS OBJECT (  eventname VARCHAR2(64),  details VARCHAR2(512),  username VARCHAR2(30),  timestamp DATE,  STATIC FUNCTION make (  details\_in IN VARCHAR2) RETURN Event\_t  );  /  CREATE OR REPLACE TYPE BODY Event\_t  AS  STATIC FUNCTION make (  details\_in IN VARCHAR2) RETURN Event\_t  IS  BEGIN  RETURN Event\_t(  SYSEVENT,  details\_in,  LOGIN\_USER,  SYSDATE);  END;  END;  /  The RETURN statement relies on two of the event-related functions, SYSEVENT and LOGIN\_USER, to record characteristics at the time of login.  Run the *aq.sql* script to create all elements and then test the code by spawning a second SQL\*Plus session to watch or dequeue the login messages.  **6.2.1.3 Trapping system errors**  The SERVERERROR event will prove to be a very handy mechanism. You can define it at the database level, which means that any error raised in any schema will be interceptable through the trigger. You can also define a trigger for this event at the schema level, limiting the scope of firing of the trigger.  Suppose that you simply want to keep track of errors raised in a particular application running on your instance. When an error occurs, you write a message to a database pipe. Another session (running asynchronously to the application users) can then wake up and dump the contents of the pipe and examine the errors.  To facilitate that process, I have created and included on the disk a package called watch. Stored in *watch.pkg* on the companion disk, this package allows you to watch actions and then direct a message constructed for that action to either the screen via DBMS\_OUTPUT or to a pipe via DBMS\_PIPE.  **TIP:** If you are comfortable with Oracle Advanced Queuing, you'll find that the AQ facility certainly offers a more robust architecture than the basic database pipes of DBMS\_PIPE for intersession communication and logging. DBMS\_PIPE, on the other hand, is easier to use and also very handy to know for a variety of development scenarios.  Using the watch package, I first create a utility procedure that I will call in my SERVERERROR triggers:  /\* Filename on companion disk: serverr.trg \*/  CREATE OR REPLACE PROCEDURE pipe\_error (  context IN VARCHAR2, msg IN VARCHAR2)  IS  BEGIN  /\* Send the information to a pipe. \*/  watch.topipe;    /\* Retrieve all system event attributes. \*/  watch.action (context || ' trap\_error', msg);  END;  /  I can then define a trigger at the database level that displays all of the attributes available from within this trigger:  CREATE OR REPLACE TRIGGER trap\_error  AFTER SERVERERROR ON DATABASE  BEGIN  pipe\_error ('DATABASE',  sysevent || '-' ||  instance\_num || '-' ||  database\_name || '-' ||  SQLCODE || '-' ||  server\_error (1) || '-' ||  login\_user  );  END;  /  I will also define a SERVERERROR trigger for the SCOTT schema, so we can explore the way multiple triggers of the same type fire:  CREATE OR REPLACE TRIGGER scott\_trap\_error  AFTER SERVERERROR ON SCOTT.SCHEMA  BEGIN  pipe\_error (login\_user,  sysevent || '-' ||  instance\_num || '-' ||  database\_name || '-' ||  SQLCODE || '-' ||  server\_error (1) || '-' ||  login\_user  );  END;  /  To test these triggers, I created the following script:  /\* Filename on companion disk: serverr.tst \*/  DECLARE  exc EXCEPTION;  PRAGMA EXCEPTION\_INIT (exc, -&1);  BEGIN  RAISE exc;  END;  /  Now I will connect as SCOTT and run the script emulating a date-related error:  SQL> @serverr.tst 1855  \*  ERROR at line 1:  ORA-01855: AM/A.M. or PM/P.M. required  If the triggers fired, the watch pipe should contain some information. I can dump the contents of the pipe with a call to watch.show:  SQL> exec watch.show  Contents of WATCH Trace:  \*\*\*WATCHing at: June 1, 1999 12:54:14  Context: SCOTT trap\_error  Message: SERVERERROR-1-ORACLE-0-1855-SCOTT  \*\*\*WATCHing at: June 1, 1999 12:54:14  Context: DATABASE trap\_error  Message: SERVERERROR-1-ORACLE-0-1855-SCOTT  We learn a few things from this execution:   * The schema-level trigger fires before the database-level trigger.   **TIP:** Current behavior in the Oracle database is that a schema-level trigger will fire before a database-level trigger. This sequence is not, however, a documented feature and is subject to change in future versions of Oracle. You should design your triggers so that the trigger logic does *not* depend on the firing sequence.   * By the time the SERVERERROR trigger fires, the SQLCODE function does not return the error that was raised. Instead, it returns 0. You must rely on the SERVER\_ERROR function to retrieve the error at the top of the error stack.   Now I will connect to the DEMO account in my Oracle 8.1.5 instance and run this script for a different error:  SQL> connect demo/demo.  SQL> @serverr.tst 1652  \*  ERROR at line 1:  ORA-01652: unable to extend temp segment by  *nnn*  in tablespace  When I take a look at my pipe contents, I have only one entry:  SQL> exec watch.show  Contents of WATCH Trace:  \*\*\*WATCHing at: June 1, 1999 13:01:38  Context: DATABASE trap\_error  Message: SERVERERROR-1-ORACLE-0-1652-DEMO  And that is because I did not create a SERVERERROR trigger in the DEMO schema; only the database-level trigger is fired.  **6.2.1.4 Checking for specific errors**  One other useful technique in this type of trigger is to check for a specific error and then take special action in that instance. There are two ways to do this:   * Use the trigger WHEN clause to restrict the execution of the trigger logic for a specific error. * Use the IS\_SERVERERROR function to determine if a specific error has been raised anywhere within the error call stack and then take action.   Let's see how each approach would work. First, suppose that I want to qualify my SERVERERROR trigger at the database level to fire only when the ORA-02292 error occurs ("integrity constraint ( *constant name* ) violated - child record found").  I can create my trigger as follows:  /\* Filename on companion disk: serverr2.trg \*/  CREATE OR REPLACE TRIGGER scott\_trap\_parent\_key  AFTER SERVERERROR ON DATABASE  WHEN (SYS.SERVER\_ERROR(1) = 2292)  BEGIN  pipe\_error (  'DATABASE trap parent key',  'Invalid attempt to delete primary key by ' ||  LOGIN\_USER);  END;  /  My call to SYS.SERVER\_ERROR(1) retrieves the error at the top of the error stack. There are two things to note in this procedure:   * If I do not qualify the call to SERVER\_ERROR with its owner, SYS, I get the following error:   AFTER SERVERERROR ON DATABASE  \*  ERROR at line 2:  ORA-00942: table or view does not exist   * I must specify the error number as a positive, not negative, integer value. Otherwise, no match will be detected.   Once the trigger is defined, I test it by trying to delete a row from the dept table whose parent key is referenced by an employee:  SQL> DELETE FROM dept WHERE deptno=10;  \*  ERROR at line 1:  ORA-02292: integrity constraint (SCOTT.FK\_DEPTNO) violated -  child record found  I can now see what information was sent to my pipe within the trigger as follows:  SQL> exec watch.show  Contents of WATCH Trace:  \*\*\*WATCHing at: June 2, 1999 16:27:09  Context: DATABASE trap parent key trap\_error  Message: Invalid attempt to delete primary key by SCOTT  Here I will demonstrate the IS\_SERVERERROR approach with a modified version of the database-level trigger:  /\* Filename on companion disk: serverr.trg \*/  CREATE OR REPLACE TRIGGER trap\_error  AFTER SERVERERROR ON DATABASE  BEGIN  /\* Same tracking as before. \*/  pipe\_error ('DATABASE', ...);    IF IS\_SERVERERROR (1652) -- POSITIVE NUMBER REQUIRED!  THEN  /\* Add a file to the tablespace... just a dummy  entry for the book. \*/  pipe\_error ('DATABASE', 'Add to tablespace');  END IF;  END;  /  And now if the ORA-01652 error is raised *anywhere* in the error stack from the SCOTT schema, I see three entries in the error pipe:  SQL> DECLARE  2 exc EXCEPTION;  3 PRAGMA EXCEPTION\_INIT (exc, -1652);  4 BEGIN  5 RAISE exc;  6 EXCEPTION  7 WHEN OTHERS THEN  8 RAISE VALUE\_ERROR;  9 END;  10 /  \*  ERROR at line 1:  ORA-06502: PL/SQL: numeric or value error  ORA-01652: unable to extend temp segment by  *nnn*  in tablespace  SQL> exec watch.show  Contents of WATCH Trace:  \*\*\*WATCHing at: June 3, 1999 09:47:05  Context: SCOTT trap\_error  Message: SERVERERROR-1-ORACLE-0-6502-SCOTT  \*\*\*WATCHing at: June 3, 1999 09:47:05  Context: DATABASE trap\_error  Message: SERVERERROR-1-ORACLE-0-6502-SCOTT  \*\*\*WATCHing at: June 3, 1999 09:47:05  Context: DATABASE trap\_error  Message: Add to tablespace  **WARNING:** When you use IS\_SERVERERROR, you must pass a *positive* number for it to find a match on the error. So we must wonder once again: are Oracle error numbers negative or positive? It looks like the folks at Oracle have trouble giving a consistent answer to this question.   |  |  |  | | --- | --- | --- | | [Previous: 6.1 Triggers on Nested Table View Columns](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_01.htm) | [Oracle PL/SQL Programming Guide to Oracle 8i Features](http://docstore.mik.ua/orelly/oracle/guide8i/index.htm) | [Next: 6.3 Schema-Level Event Triggers](http://docstore.mik.ua/orelly/oracle/guide8i/ch06_03.htm) | | 6.1 Triggers on Nested Table View Columns | [Book Index](http://docstore.mik.ua/orelly/oracle/guide8i/index/idx_0.htm) | 6.3 Schema-Level Event Triggers |   The Oracle Library Navigation  [Copyright (c) 2000](http://docstore.mik.ua/orelly/oracle/copyrght.htm) O'Reilly & Associates. All rights reserved. |
|  |

|  |
| --- |
| [bigmir)net](http://www.bigmir.net/)  4805  8326 |

[http://counter.rambler.ru/top100.cnt?692798](http://top100.rambler.ru/top/)[Ramblers Top100](http://top100.rambler.ru/top100/)