HANDS-ON PRACTICE 4
LOADING AUDIT COLUMNS

Peter Koletzke, Quovera
Duncan Mills, Oracle Corp.

Now we will look at a different approach to the same problem implemented within the database itself. As we mentioned before, you may choose to take this approach if you have other applications accessing the same data or requirements that are not covered by the basic history column functionality of ADF BC.

Implementing audit columns or a journal table in PL/SQL requires BEFORE row-level triggers for each operation. The row-level trigger updates an audit column or inserts the values of the old (or new) row in the journal table; alternatively, the journal table could hold just the data that has been changed. The problem is that if your application uses a single database user account to connect to the database (as do most J2EE web applications), the table triggers cannot use the USER pseudo-column to assign the created or modified user columns. For example, in an Oracle Forms application where the user logs in using his or her own database account, a BEFORE INSERT row-level database trigger would contain the following:

```
:NEW.created_by := USER;
:NEW.created_date := SYSDATE;
```

You can create named values (like variables) that you store in the application context by assigning a value to a name. You can then read the values by accessing the value name within that same database session. Since you declare these variables at runtime (not in explicit declarative code), context name-value pairs are akin to global variables in Oracle Forms. In Oracle Forms, you create global variables by assigning them, and you can then read the value throughout the same database session.

The entire solution requires the following elements:

- **An application context** You access a context by a name; the same context can provide session-specific information for many database sessions, although the values assigned into the context in one session will be readable only within that session. Therefore, you only need one application context for this purpose.
- **A PL/SQL context package** This package contains procedures and functions that write and read to the context area.
- **J2EE code to write to the context** The J2EE code will call a procedure in the context package to write a value for the logged-in user name to the application context.
- **Table trigger code to read the context** The built-in database function SYS_CONTEXT can access the context in the trigger. For example, you could create a context called HR_CONTEXT; application code would write a value called APP_USERNAME into that context with the logged-in user name. The snippet shown earlier from the BEFORE INSERT table trigger would then become this:

  ```
  :NEW.created_by := SYS_CONTEXT('HR_CONTEXT', 'APP_USERNAME');
  :NEW.created_date := SYSDATE;
  ```

The following practice is taken in part from the *Oracle JDeveloper 10g for Forms and PL/SQL Developers* (Oracle Press, McGraw-Hill/Osborne). The technique consists of the following phases:

I. Create the database objects

II. Set the context from the application

- Call the database procedure from prepareSession()
- Test the method call
- Move the code to the framework class

> **Note**
>
> Code snippets are available in text files that accompany this practice.

I. Create the Database Objects

Although this section describes how to implement the audit column technique, you could apply many of the same principles to inserting a row into a journal table. Although the PL/SQL triggers and tables will differ between the two requirements, the problem is the same—how to access the logged-in user name from within PL/SQL code.

The key to the solution is an application context, an Oracle database feature that allows you to store and query information for a specific user session. You need to create the context and associate it with a PL/SQL package that will manage writing and reading from the context. You can think of the application context as a memory area, which you can write to and read from throughout a user's database session.

You can create named values (like variables) that you store in the application context by assigning a value to a name. You can then read the values by accessing the value name within that same database session. Since you declare these variables at runtime (not in explicit declarative code), context name-value pairs are akin to global variables in Oracle Forms. In Oracle Forms, you create global variables by assigning them, and you can then read the value throughout the same database session.

The entire solution requires the following elements:

- **An application context** You access a context by a name; the same context can provide session-specific information for many database sessions, although the values assigned into the context in one session will be readable only within that session. Therefore, you only need one application context for this purpose.
- **A PL/SQL context package** This package contains procedures and functions that write and read to the context area.
- **J2EE code to write to the context** The J2EE code will call a procedure in the context package to write a value for the logged-in user name to the application context.
- **Table trigger code to read the context** The built-in database function SYS_CONTEXT can access the context in the trigger. For example, you could create a context called HR_CONTEXT; application code would write a value called APP_USERNAME into that context with the logged-in user name. The snippet shown earlier from the BEFORE INSERT table trigger would then become this:

```
:NEW.created_by := SYS_CONTEXT('HR_CONTEXT', 'APP_USERNAME');
:NEW.created_date := SYSDATE;
```

You need to follow the steps in this section to make the required database changes.

2. Log in to the database as SYS, or a DBA account connected as sysdba, and grant access to the application object owner (HR) using the following statements:

```
GRANT EXECUTE ON SYS.dbms_session TO hr;
GRANT CREATE ANY CONTEXT TO hr;
GRANT DROP ANY CONTEXT TO hr;
GRANT SELECT ON SYS.v_$session TO hr;
```

3. Log in as HR, and add audit columns to the table. This example uses the EMPLOYEES table, so you would run the following:

```
ALTER TABLE employees
ADD (
   created_by     VARCHAR2(30),
   created_date   DATE,
   modified_by    VARCHAR2(30),
   modified_date  DATE);
```

4. Update the CREATED_BY and CREATED_DATE columns with appropriate values, and change those columns to NOT NULL, as follows:

```
UPDATE employees
SET created_by = 'HR',
    created_date = TO_DATE('01/01/1980 12:12','MM/DD/YYYY HH24:MI');

ALTER TABLE employees
MODIFY ( created_by NOT NULL,
         created_date NOT NULL);
```

5. Create the context and refer to a package (even though this package is not yet created):

```
CREATE CONTEXT hr_context USING security_pkg;
```

6. Create the SECURITY_PKG using the following SQL:

```
CREATE OR REPLACE PACKAGE security_pkg
IS
   PROCEDURE set_security_context (
      p_username    IN VARCHAR2,
      p_application IN VARCHAR2 DEFAULT 'TUHRA');

   FUNCTION logged_in_user
      RETURN VARCHAR2;

END security_pkg;

CREATE OR REPLACE PACKAGE BODY security_pkg
IS
   PROCEDURE set_security_context (
      p_username    IN VARCHAR2,
      p_application IN VARCHAR2 DEFAULT 'TUHRA')
   IS
   BEGIN
      -- Write the user info into the context area
      -- The application name is used later
      SYS.DBMS_SESSION.set_context ('HR_CONTEXT', 'APP_USERNAME', p_username);
   EXCEPTION
      WHEN OTHERS
      THEN RAISE_APPLICATION_ERROR(-20001,
            'Error in SECURITY_PKG.SET_SECURITY_CONTEXT: ' || SQLERRM);
   END;

   FUNCTION logged_in_user
      RETURN VARCHAR2
   IS
      v_username   VARCHAR2(100);
   BEGIN
      v_username := UPPER(SYS_CONTEXT('HR_CONTEXT', 'APP_USERNAME'));
      RETURN v_username;
   EXCEPTION
      WHEN OTHERS
      THEN RETURN 'Error in LOGGED_IN_USER';
   END;

END security_pkg;
```

Additional Information: The Java code you write later will pass the logged-in user name to the set_security_context procedure. It will also pass the application name. Although the code just shown does not use this parameter, the section “What Could You Do Next?” later in this practice explains its possible use.

7. Create the trigger to set the audit columns as follows:

```
CREATE OR REPLACE TRIGGER employees_audit_biu
   BEFORE INSERT OR UPDATE
   ON employees
   FOR EACH ROW
DECLARE
   v_user   VARCHAR2(30);
BEGIN
   v_user := security_pkg.logged_in_user;

   IF INSERTING
   THEN
      :NEW.created_by := v_user;
```

```
      :NEW.created_date := SYSDATE;
   ELSIF UPDATING
   THEN
      :NEW.modified_by := v_user;
      :NEW.modified_date := SYSDATE;
   END IF;
END;
/
```

8. You can test your code in the SQL tool of your choice (for example, JDeveloper's SQL Worksheet). First note the contents of the audit columns for a particular row in the EMPLOYEES table:

```
SELECT employee_id,
       first_name, last_name,
       created_by,
       TO_CHAR(created_date, 'mm/dd/yyyy hh24:mi:ss') created_date,
       modified_by,
       TO_CHAR(modified_date, 'mm/dd/yyyy hh24:mi:ss') modified_date
FROM   employees
WHERE  employee_id = 100;
```

9. Set the application context parameter for the session as follows:

```
BEGIN
   security_pkg.set_security_context('TFOX', 'TUHRA');
END;
```

10. Check that the application context parameter is set:

```
SELECT security_pkg.logged_in_user
FROM dual;
```

11. Issue an UPDATE statement. The following does not change anything, but it will cause the trigger to fire:

```
UPDATE employees
SET    last_name = last_name
WHERE  employee_id = 100;
```

12. Check the audit column values by re-running the SELECT statement from step 7. You should see a MODIFIED_BY value of "TFOX" and the MODIFIED_DATE of today.

What Did You Just Do?

You added audit columns to the EMPLOYEES table and created an application context to hold the value of the user name, which will be passed to the application context. You also created a package associated with the application context. This package contains a procedure to add the user name to the application context and a function to read the user name from the context. In addition, you created a trigger on the EMPLOYEES table that loads the user name into an audit column during an INSERT or UPDATE. Finally, you tested these database components by updating the EMPLOYEES table and viewing the resulting audit column values.

What Could You Do Next?

Naturally, you can add this kind of functionality to more than one table. Each table requires audit columns and a trigger to load the user name and dates for INSERT and UPDATE operations. The package and application context would be shared by all these triggers.

You could use a similar technique to implement a journal table system by creating a duplicate of the main table (for example, EMPLOYEES_ JN), with additional columns for OPERATION ("INSERT" or "UPDATE"), OPERATION_DATE, and OPERATION_USER. The trigger would be modified to insert a record with the old values and values for the additional columns into this table. Again, the package and application context would be the same as before.

You might want to add a condition to the audit columns trigger so that if a user is not logged in through the web application, the trigger will record the database login user name instead of the web application login name. This requires the following changes:

- **An additional context value** IS_WEB_USER that you assign as "Y" in the set_security_context function.
- **Additional trigger logic** to check if the context variable IS_WEB_USER is set to "Y." If so, the user name column is set to the user name in the context. Otherwise, it is set to USER (the logged-in database user).

Since all web users log in as the same database user, there is no way to distinguish their sessions when viewing the virtual view, V$SESSION. However, V$SESSION contains two columns, CLIENT_INFO and ACTION, into which you can write information such as the logged-in user name. To add user-specific information to these columns, you would add these two lines to the security_pkg.set_security_context procedure:

```
DBMS_APPLICATION_INFO.SET_CLIENT_INFO(p_username);
DBMS_APPLICATION_INFO.SET_ACTION(p_application || ' - ' ||
       TO_CHAR(SYSDATE,'MM/DD/YYYY HH24:MI'));
```

You must also grant the connection user EXECUTE privilege on DBMS_APPLICATION. The first line writes the user name into the CLIENT_INFO column, and the second line writes the value of the p_application parameter (set to "TUHRA" by the application code) and the date and time into the ACTION column.

> **Note**
>
> V$SESSION displays information about database connections. Since a J2EE web application has no persistent database connection and since ADF BC pools database connections, you do not know the state of user sessions that appear to be connected in V$SESSION. For example, a user name may appear in V$SESSION even though the user has closed the browser, because the database connection in the application module pool may not have been reused.

II. Set the Context from the Application

Now that the PL/SQL and database objects are in place, all that remains is to write code in the application to call the database function that sets the application context value for the user name. This phase uses a technique to call a database procedure and to define code that will be executed automatically as the ADF Business Components layer establishes the user session with the database.

Call the Database Procedure from prepareSession( )

This section extends the functionality of the application module prepareSession() method, which the ADF BC will automatically call as the database connection is established (for each interaction with the database). The advantages of adding the context initialization code to this method follow:

- The framework will execute the context setting code for you. You do not need to explicitly execute the call from the user interface code.
- The prepareSession() method will automatically be reinvoked and the context set correctly if the application employs connection pooling or application module pooling,
- The context will be reset with the new ID if the user logs out of the application and connects using another ID.

The following steps start with the sample TUHRA (The Ultimate Human Resources Application) application workspace. In addition to the database code described in Phase I of this practice, they require an Employees Edit page and the security features worked into the application in the first practice.

1. Select the TuhraService application module node under Model\Application Sources\tuhra.model. Double click TuhraServiceImpl.java in the Structure window to open the Java file in the editor.
2. Select Source | Override Methods and select the checkbox for the prepareSession(oracle.jbo.Session) method. (This method will display an open lock icon, indicating that it is public.) Click OK. The method code stub will be inserted into the code file.
3. Modify the generated prepareSession() method so it includes the following code:

```
public void prepareSession(Session session)
{
  super.prepareSession(session);
  // Retrieve the J2EE user ID
  String authenticatedUser = getUserPrincipalName();

  if ( (authenticatedUser != null) &&
       (authenticatedUser.trim().length() > 0))
  {
    DBTransactionImpl dbTransaction = (DBTransactionImpl)getDBTransaction();
    // Parameter for application name
    String pApplication = "TUHRA";
    // Transaction statement with procedure call
    CallableStatement callableStmt = dbTransaction.createCallableStatement(
      ("BEGIN " +
         "security_pkg.set_security_context(?, ?); " +
       "END;"), 0);
    try
    {
      // Register parameters and call procedure
      callableStmt.setString(1, authenticatedUser);
      callableStmt.setString(2, pApplication);
      callableStmt.execute();
    } catch (SQLException sqlExcept)
    {
      throw new JboException(sqlExcept);
    }


    {
      try
      {
        if (callableStmt != null)
        {
          callableStmt.close();
        }
      } catch (SQLException closeExcept)
      {
        throw new JboException(closeExcept);
      }
    }
  }
}
```

Additional Information: This code uses an API function provided by the ApplicationModuleImpl superclass—getUserPrincipalName(), which returns the ID of the user that is authenticated by the J2EE container. If the user name is not null, the code then creates a database transaction object, dbTransaction, and prepares a statement with two replaceable parameters (user name and application name) to call the database procedure. Next, the code registers the parameters and executes the PL/SQL statement.

4. You will see many class names with wavy underlines. These indicate missing imports. Hold the mouse cursor over DBTransactionImpl, and press ALT-ENTER to add the import (oracle.jbo.server.DBTransactionImpl). Repeat this operation for CallableStatement.
5. Hold the mouse cursor over one of the callableStmt lines, and import SQLException. Repeat this for JboException (select oracle.jbo.JboException from the pulldown).
6. Compile the code (using Make from the right-click menu in the editor). Fix any problems and recompile.
7. Click Save All. The method is now defined and will execute automatically when a new session is created.

Test the Method Call

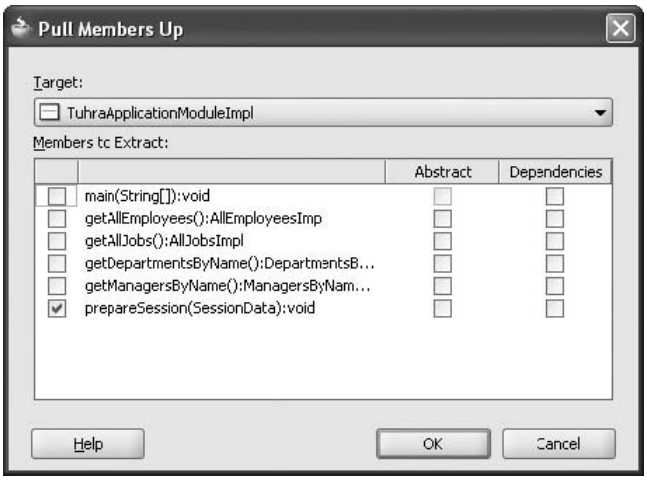
Now that the setup is complete, you can run a test in the application.

1. Open the Table Viewer in JDeveloper for the EMPLOYEES table (in the Connections Navigator, open the Database, HR, HR, and Tables nodes; and double click the EMPLOYEES node). Select the Data tab.
2. Select an employee record, and note the values for the columns MODIFIED_BY and MODIFIED_DATE (probably NULL).
3. Return to the Applications tab, and run the application. Log in as TFOX. On the Employees tab, query and edit the employee record you noted in the preceding step. Change the phone number, and click Save.
4. Close the Table Viewer in JDeveloper, and reopen it. The MODIFIED_BY and MODIFIED_DATE for the record should now be TFOX and the current date.
5. Log out and log in again as NKOCHHAR. Update the same record, and note the change in the audit column values.
6. Close the browser and stop the server.

Move the Code to the Framework Class

Although this application only contains one application module at this stage of development, you expect the project to grow so that it contains more than one application module. You will need to apply the same context-setting procedure call to all application modules. Instead of repeating the method in each application module, you can code it on the application framework level so that one method is available to all application modules. In this section, you will move the method call to the framework code level.

1. Open the TuhraServiceImpl.java file if it is closed. (Select the application module and double click the file name in the Structure window.)
2. Place the cursor in the prepareSession() method, and select **Refactor | Pull Members Up**. The Pull Members Up dialog will appear, as shown here:



3. Be sure the prepareSession() method checkbox is selected.
4. Notice the Target pulldown contains the name of the superclass TuhraApplicationModuleImpl. Click OK. The method will disappear from the editor.
5. Scroll to the class declaration line (public class TuhraServiceImpl) and, on the TuhraApplicationModuleImpl class name (in the extends TuhraApplicationModuleImpl clause), select Go to Declaration from the right-click menu. The superclass file will open in the editor. Verify that prepareSession() was moved. This method will now be available to all application modules in the same project.
6. Click Save All.
7. Run the application. Log in as TFOX again. Make a change to a record, and check that the audit columns are updated.
8. Close the browser and stop the server.

What Did You Just Do?

You wrote code in the application layer to call the security_pkg.set_security_context procedure so the logged-in user name would be placed in the application context and the INSERT and UPDATE trigger could write the proper user name into the table's audit columns. You did this by overriding and adding functionality to a framework method, which is called as the database session is established. Then you tested this code and moved the method to the framework class so that all application modules in the same project will be able to take advantage of it.

> **Tip**
>
> Should you wish to share this code among projects, you can create a cross-application framework file that extends the ADF class. The new cross-application framework file would be distributed in a library that you attach to each Model project you create. Your project-level framework class would then subclass the cross-application framework class.

You could also implement Virtual Private Database (VPD) features that restrict a user's access to specific rows. The technique just described writes the logged-in user name to the application context. You can write functions that return WHERE clause predicates using the user name in the context to restrict rows. You then attach the functions to tables by setting up policies, definitions that cause a policy function to be executed when any SELECT statement is issued to its attached table. More information about VPD can be found in the Oracle Application Developer's Guide–Fundamentals, available in the Oracle database documentation.