

TECHNIQUES FOR DEVELOPING DATABASE API'S WITH PL/SQL PACKAGES

John C. Lennon, Dulcian Inc.

Introduction

Since the release of Oracle7, the Oracle database has supported multi-tier environments. The introduction of stored functions and procedures allowed us to begin to consider more efficiently modularizing our applications. Code that addressed the database could be moved from the client onto the database server. The advantages of doing this were not immediately apparent; it just looked like an additional complication. The lack of documentation and information available resulted in a slow acceptance of what is now a widely used powerful and valuable addition to Oracle's toolset. What was once seen merely as a method for improving performance in the client/server environment is now viewed as an essential ingredient for successfully developing applications for the intranet/internet with new features being added with each new Oracle release. This paper concentrates on the 2 tier client/server environment, however what is described here applies equally to 3 tier applications.

The Methodology

The development team with which I was working six years ago can be described as early adopters. As mentioned in the introduction to this paper very little information was readily available, but, having experimented for a while it was decided that using PL/SQL packages was the way forward. The application being developed called for very rapid response times, and we had began to recognize that we had some very serious performance bottlenecks. These would need to be addressed if we were going to satisfy our client's requirements. We were using Designer/2000 as a development tool and it soon became apparent that the forms generator, as it then existed, was not going to support our requirements. The code that was generated could not, in most cases, be migrated to the server without a considerable amount of re-work, which would defeat the object of generating forms. At first we were reluctant to abandon using the forms generator, especially as significant time and effort had gone into creating templates and setting up preferences. Eventually, though, we realized that we could not have it both ways and the decision was made to create forms manually using Developer/2000 (Forms 4.5). Predictably, this shortcoming has been addressed by the Oracle Designer and Developer groups and both products now support the methodology.

To take full advantage of the performance enhancements available using stored functions and procedures in packages, we opted for what was then a radical choice. With the exception of block base table operations and populating record groups for lists of values, no SQL would be permitted on the client, neither in forms nor libraries. SQL would be stored and executed only on the database server. One concern, though, was managing the myriad of objects we realized we would need to create in the database. One of the advantages of using packages is that they are an ideal way to group together PL/SQL elements that are in some way related to one another. This means that there are a lot fewer objects that have to be managed. There are other advantages to packages, particularly on the database server, and these are discussed later.

We decided that, in the database, we would create four types or classes of package.

- **The Application Package**
This contains all elements that are used to control the behavior of the application and elements that are used by numerous modules. (This may have to be split into more than one package if it grows too large.)
- **Library Packages**
These contain elements used by client or web server PL/SQL libraries.
- **Table Packages**
These contain the elements that serve to encapsulate a table.
- **Forms Packages**
These contain elements that are used by only one form and that require access to database objects.

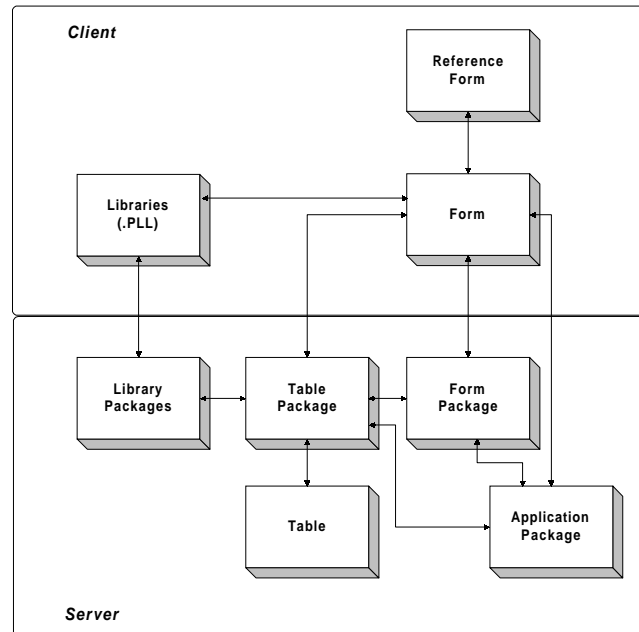


Figure 1

These packages communicate between each other and the client side modules as shown in *Figure 1*. For most purposes the application package(s) and the library packages could be considered one and the same. They have been separated for practical reasons connected with size and the inherent instability of Developer/2000 libraries.

By taking the decision to move away from using Designer to generate modules we were obviously sacrificing the advantages it offers in developing modules rapidly, with little or no coding, and with a consistent GUI look and feel. To continue to meet the deadlines mentioned earlier, it was necessary to provide the infrastructure to support our developers and enable them to concentrate their efforts on the functionality of their forms. To maintain conformity with GUI standards, we developed a standard reference form in which are defined all GUI objects needed for the application. A consistent look and feel is ensured through standard alerts, property classes and visual attributes referenced by the developers' forms.

Standard triggers were also developed. Some of these, like pre-block and post-block triggers, are concerned with maintaining GUI standards. Others, like when-new-block, when-new-record and when-new-item triggers, control the toolbar and menu, enabling and disabling buttons and menu items based on block, record and item properties. Additional referenced triggers, including pre-insert, pre-update and pre-delete triggers, save the developer the drudgery of writing repetitive code. Such mundane things as populating audit columns, fetching sequences, verifying referential integrity and displaying a calendar as a list of values for a date item are handled in the background.

Notwithstanding this built-in functionality, there is no SQL in any of the form triggers, only calls to functions and procedures in libraries. In this way, any necessary changes can be made without the need to re-generate any forms. With a few exceptions, the elements within these libraries are contained in packages. The exceptions are mainly historic and have to do with backward compatibility. They would otherwise be placed in packages.

Setting up the environment is quite time consuming, but it has been our experience that, within the framework we have provided, a form module can be completed in about the same time as it takes to design, define and generate using Designer. On the downside, the documentation that would be provided by Designer is not available and until recently packages could not be satisfactorily reverse engineered.

Client Side Packages

The principal advantage of packages on the client or application server is that they provide a structure within which to organize other PL/SQL elements. Packages provide a context and structure for related elements. (There is no performance gain; that comes only with database packages.) Another advantage is being able to use persistent objects such as variables and constants. These can either be public and available to all other elements within the current form, or private and available only to elements within the same package.

A good example of a client side package is the one we use to control our forms toolbar. Defined within a library, the package is called, unsurprisingly, "toolbar." Declared in the package specification are procedures such as `Button_Pressed` and `Disable_Button`. The toolbar has been designed to be completely generic. Buttons may or may not be displayed in different forms, and sometimes buttons in the same position have a different functionality and therefore different icons. The buttons as they appear in the form are merely numbered. Variables declared in the package specification are used to assign names and icons to buttons, as shown in *Figure 2*.

```

PACKAGE Toolbar
IS
  Save_Button VARCHAR2(64) := 'Toolbar.Button1';
  Exit_Button VARCHAR2(64) := 'Toolbar.Button2';
  .
  .
  .

  Save_Icon VARCHAR2(8) := 'save'; -- save.ico
  Exit_Icon VARCHAR2(8) := 'exit'; -- exit.ico
  .
  .
  .

  PROCEDURE Button_Pressed;
  PROCEDURE Disable_Button(p_Button_Name IN VARCHAR2);
  .
  .
  .

END Toolbar;

```

Figure 2

The developer refers to the buttons using variables - which are in effect synonyms - without concern for the physical position of the button. For example, the developer refers to `Toolbar.Disable_Button(Toolbar.Save_Button)` rather than to `Toolbar.Disable_Button('toolbar.button1')`. The developer also needs to be concerned only with what is visible in the package specification. The underlying implementation has been hidden from view or "black boxed."

Packages can contain almost any type of PL/SQL element. However, in keeping with our philosophy of no SQL on the client, for client-side packages, elements that access the database, such as cursors, are called from the stored library package.

Overloaded Names

A powerful feature of packages is support for the use of overloaded names. This means that more than one function or procedure with the same name can exist within a package, provided the parameter lists or, in the case of a function, the return data type, of each are different. The parameters can differ in data type, number, or both, provided the combination is unique. When the function or procedure is called, SQL or PL/SQL determines which to use based on the data type and number of the parameters being passed.

We have used this to good effect for some of the utilities provided for our developers. They do not have to decide which one of a number of procedures to call, only which parameters to pass.

```

v_Message := Message_Pkg.Get_Message(195);
set_alert_property('error_alert'
                 ,message_text
                 ,v_Message);
show_alert('error_alert');
RAISE form_trigger_failure;

User_Alert.Display(195);

```

Figure 3

An illustration of this is a package named `User_Alert`. This was written to eliminate repetitive and tedious coding, as demonstrated in *Figure 3*.

In this example, the procedure calls an alert named `error_alert` sets the text of the alert message to message number 195 from the `MESSAGE` table and finally raises `form_trigger_failure`. This is exactly what the preceding four lines of code did. This is the most commonly used of four overloaded procedures named `Display`. These allow for using a hard-coded test message, displaying a different alert, displaying the message on the console message line instead of an alert and choosing whether or not to raise `form_trigger_failure`. This is the most commonly used of four overloaded procedures named `Display`. These allow for using a hard-coded test message, displaying a different alert, displaying the message on the console message line instead of an alert and choosing whether or not to raise `form_trigger_failure`. The `User_Alert` package is shown in *Figure 4*.

Stored Packages

In addition to the advantages already described, packages stored in the database offer very real performance gains. The first and most obvious is through a significant reduction in network traffic. The less obvious, but even more significant gain, results from how Oracle implements stored packages in the database.

When any object in a package is first referenced, the entire package is compiled, validated and loaded into memory in the shared global area (SGA). The whole package is thus made available for future calls to any of its elements. This applies not just to the current session but also to all other sessions. The package remains in the SGA indefinitely unless it is removed by the least used algorithm to free up memory. PL/SQL does not have to retrieve elements from the disc when they are called. It is possible to "pin" a package into the SGA. However, although the means to do this have been provided, this practice is not normally recommended. Only rarely used packages are likely to be affected, and packages that fall into this category are unlikely to have a significant effect on the overall performance of the application.

Further performance gains are realized by taking advantage of variables and constants declared in a package specification, and therefore public. These can be used to store frequently used values to avoid continually retrieving them. Alternatively, if a frequently used variable needs to be protected against unauthorized alteration, it could be declared privately in a package body and a public function created in the package to retrieve the value. An example of this is storing a user ID. In our application we follow the Oracle Applications model and use audit columns that are populated with the date, time and user ID on insert and update. The user ID is not the Oracle username but an

```

PACKAGE User_Alert
IS
  PROCEDURE Display(p_Message          IN VARCHAR2
                  ,p_Alert              IN VARCHAR2
                  ,p_Raise_Form_Fail    IN BOOLEAN);
  PROCEDURE Display(p_Message          IN NUMBER
                  ,p_Alert              IN VARCHAR2
                  ,p_Raise_Form_Fail    IN BOOLEAN);
  PROCEDURE Display(p_Message IN VARCHAR2);
  PROCEDURE Display(p_Message IN NUMBER);
END Show_Alert;

PACKAGE BODY User_Alert
IS
  PROCEDURE Display(p_Message          IN VARCHAR2
                  ,p_Alert              IN VARCHAR2
                  ,p_Raise_Form_Fail    IN BOOLEAN)
  IS
    v_Alert VARCHAR2(16) := 'error_alert';
  BEGIN
    IF upper(p_Alert) = 'I' THEN
      message(p_Message);
    ELSE
      IF upper(p_Alert) = 'F' THEN
        v_Alert := 'fatal_alert';
      END IF;
      set_alert_property(v_Alert
                       ,message_text
                       ,p_Message);
      show_alert(v_Alert);
      IF p_Raise_Form_Fail THEN
        RAISE form_trigger_failure;
      END IF;
    END IF;
  END Display;
  PROCEDURE Display(p_Message          IN NUMBER
                  ,p_Alert              IN VARCHAR2
                  ,p_Raise_Form_Fail    IN BOOLEAN)
  IS
  BEGIN
    Display(Message_Pkg.get_Message(p_Message)
           ,p_Alert
           ,p_Raise_Form_Fail);
  END Display;
  PROCEDURE Display(p_Message IN VARCHAR2)
  IS
  BEGIN
    Display(p_Message
           ,'E'
           ,TRUE);
  END Display;
  PROCEDURE Display(p_Message IN NUMBER)
  IS
  BEGIN
    Display(Message_Pkg.Get_Message(p_Message)
           ,'E'
           ,TRUE);
  END Display;
END Show_Alert;

```

Figure 4

application-specific unique identifier obtained from a table. To ensure the integrity of the audit columns, even if inserts and updates are done outside of the application, database triggers are used. This implies that the USER table needs to be queried to obtain the user ID for each insert or update. To avoid this, the first time a table insert or update occurs in a session, a private variable in the application package body is populated with the user ID. A function is called by the database triggers from that point onward to read the value from the variable. This method also solves a mutating table problem if the standard triggers are implemented for the USER table.

We have used this technique for a number of other purposes. To our developers, the function looks and behaves just like a constant. Any attempt to alter its value will result in an error. Packages also provide a means of isolating data and can be used to enforce security. In a similar way to a view, a package can be owned by the owner of a table. Execute privilege can then be granted to users, allowing them restricted access to a table for which they have not been granted privileges. If necessary, security based on roles can also be applied to individual elements of a package, further enforcing security.

The Application Package

The application package is used for functions and procedures that do not directly relate to a single table and that are used by more than one module. These include procedures that are called during the application startup process, when a user first logs on to the application. These procedures establish the environment at the start of the session. The process includes assigning values to public variables declared in the application package. Values are also returned to the calling procedure and used to populate global variables in Oracle Forms. If values are to be used both by database objects and client objects, they are assigned at both levels to reduce network traffic.

The application package is also used for procedures and functions that are used to enforce complex business rules. This allows for greater flexibility and ease of maintenance.

This package also provides a container for several generic procedures that use dynamic SQL. An illustration of this is a procedure that validates tables where a date range comprises part of a unique key. This is another example of an overloaded procedure, as it can handle up to ten different combinations of the number of columns that make up the key and data-types.

Library Packages

As mentioned earlier, library packages are essentially similar to the application package but are separated from that package for practical reasons and to restrict access to library code. A library package contains all functions and procedures not directly connected with a single table. It may also contain "pass through" functions and procedures that call their counterparts in the application package and which exist only to isolate the application package from libraries.

Table Packages

Perhaps the single most important element of our methodology is table packages. A package is created for each table and most operations on tables are effected using the functions and procedures in these packages. What this achieves is to encapsulate the table and make the table and package appear to be one object, as shown in *Figure 5*.

Even with the best documentation and communication, a certain amount of code redundancy is inevitable. This method helps avoid this redundancy, because any function or procedure that interfaces with a table resides in that table's package and its location is never in doubt. Standard procedures are created for each table. These include procedures to handle inserts, updates, deletes and to retrieve records. Functions are provided to return the next sequence for the primary key, the unique key value to be used in order by clauses and, for validation, record status and values from other commonly-used columns. Some of these might appear redundant. For example, why return just one value when the whole record could be returned? The reasons are twofold. First, a single value is appropriate, such as when embedding a function in SQL, and second, one of our aims is to reduce network traffic. Returning extraneous data does not support this aim.

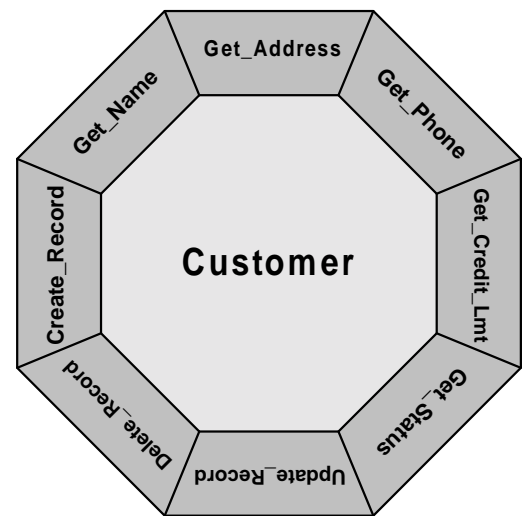


Figure 5

When a developer needs a function or procedure that is not part of the standard set, the table package is first checked to see if another developer has already created a similar procedure. If not, the new procedure is added to the table package and is immediately available to other developers.

Forms Packages

Forms packages are used when functions and procedures that are specific to a particular form are required. A typical example would be where a post-query trigger in the form needs to query multiple tables to populate foreign key lookup columns. Instead of multiple calls across the network to table packages, a procedure is placed in the form package. The form calls this procedure passing the primary key values as parameters, and the lookup values are returned as illustrated in *Figure 7*.

Another example of using a forms package is when it is necessary to update or insert into multiple tables in the background. Again, a procedure is created in the forms package that accepts the values to be updated or inserted from the form. This procedure then uses the individual table packages to complete the process.

Both of these examples achieve a considerable reduction in network traffic.

Conclusion

The methodology described here has proved very successful, and build time has proved to be comparable to that previously achieved generating modules using Designer/2000. Performance has been enhanced significantly, both by a reduction in network traffic and by executing most database transactions at the level of the database server.

The first project to use this methodology evolved into a product that is being customized for other clients. The customization effort has been greatly reduced by modularization. In many cases it has proved unnecessary to make any modification to forms modules despite significantly different business rules.

One project still under development at the present time is benefiting in particular from the reduced network traffic. With hundreds of users in three states accessing a central database, network performance is critical for success.

Forms 5 and 6 are offering support for this concept. In a number of cases, particularly where a table has a hierarchical structure, we are used packages to populate blocks. The Forms feature that provides the ability to base a block on a package instead of a table offers a solution that requires considerably less coding.

Designer 2.x and 6 have introduced the possibility of generating server side packages for those developers using this tool.

About The Author

John Lennon is a Project Manager with Dulcian Inc. Originally from England, John resided for some years in South Africa before moving to the United States in 1992. He has many years of experience with applications development, primarily in engineering and utility environments. John has been working with Oracle products for over a decade. He has presented at international and regional Oracle users' conferences and has had articles published in technical journals in the United States and Great Britain.

John C. Lennon

e-mail johnlennon@ieee.org

Web site <http://members.aol.com/jomarlen>