PL/SQL "Starter" Framework

Open Source PL/SQL Libraries for Custom Oracle Development

Introduction

An application framework is a collection of software components that implement functionality commonly needed by developers to assemble high quality software applications in a rapid, consistent manner.

The PL/SQL "Starter" Framework, or just *Starter* for short, is like a sourdough starter, or a starter engine. It is a simple, mature, robust, integrated set of pre-built libraries mean to jump start, or give a needed boost to application development, eliminating the need to design, develop and ripen these features independently.

Although this framework is simpler than others, it is still a framework and takes time to grasp everything available, and how all the pieces fit together. Everything is documented heavily in the package specifications, table and column comments, and script comments. However, pure textual documentation can only go so far. An overview is needed, something with visual cues, that ties it all together and flattens the learning curve. Hence the need for this user guide.

I assume the reader is already aware of, and a big fan of, PL/SQL, Oracle and software development best practices. Little effort will be wasted "preaching to the choir." If concepts like assertions, standardized exception handling, and instrumentation are unfamiliar, refer to the excellent books and online articles by Tom Kyte and Steven Feuerstein.

Some will be in a hurry to use a particular feature of Starter as soon as possible. For the hasty, read Getting Started, and then skip to your feature set of interest. The most typical use of Starter will be to inherit its RBAC security model and data structures. Turn to the first section, Security, for that. If your application is fragile and difficult to measure, trace and debug, turn to the Instrumentation section to see how to do that. If your application needs to use table-driven properties -- instead of hard-coded literals, or enumeration erroneously kept in the middle tier -- turn to the Configuration and Parameters section, etc. See the index of hyperlinks above to jump to the section of interest.

However, if your desire is to understand the big picture and how Starter is layered, organized and meant to be used, read on as its installation, genesis and overall design are presented.

Package Specification Comments

This document will make use of longer prose, hyperlinks, diagrams and syntax highlighted sample code to explain the topology and use of the Starter framework. None of these methods are possible in plain ASCII text.

The most in-depth documentation, covering all options and possibilities, will be maintained in the libraries' public interface, the package specification comments. When a section in this document seems a bit light on the detail, it is because I'm attempting to avoid redundancy as much as possible between these two repositories of Starter documentation.

Terminology

There are some terms used in the paper that may not be as familiar to the reader as they are to the author.

Routine = Packaged or standalone PL/SQL function or procedure

Backend = database-centric

Getting Started

Here is the shortest path to evaluate or use Starter. Visit the Installation section for greater detail.

1. Download the zip file and documentation from SourceForge (1 min.)
2. Unzip to the directory of your choice. I used C:\Dev\Core. (10 seconds)
3. Ensure there is a tablespace to contain the segments created by the framework DDL script. I created a CORE_DATA and CORE_INDEX tablespace. For you first install, it might be easier just to use the default USERS tablespace. (0 - 10 minutes)
4. Decide where to install the framework. You can use an existing account or a new one that the script will create for you. I went with a new one and named it CORE when prompted.
5. As SYSDBA, run `__InstallStarterFmwk.sql` and follow the prompts (20 seconds).

That's all there is to it. It is now installed, along with some example accounts and sample data in the tables to demonstrate how multiple applications can share the same data structures. Once the example accounts and data have served their purpose, drop them and delete them. If the example accounts were not desired in the first place, review the Installation section below for details.

6. If you want to start using the framework in an existing application, run `_create_synonyms_for_fmwk_objects.sql`. When prompted for "fmwk_consumer", enter the account name which owns PL/SQL objects that will make use of the framework.
7. Set up a few lines in APP, APP_DB, APP_ENV, APP_PARM and APP_ENV_PARM for your application (see Parameters/Configuration and the ENV package spec for further detail.
8. Explore the installed framework. Examine code in the sample app (found in the zip file). Read the sections below to get a better feel for each service offered by Starter. Then follow suit.

Genesis of Starter

This section is a short history of the Starter framework. You may skip this if you wish.

I began my career in 1995 with Andersen Consulting, transferring to regular employment in 1998. Whether I was a developer, architect, lead or tuner, the engagements and employers were always on the latest version of Oracle, had very large databases, and many backend (and sometime frontend) applications and processes written in PL/SQL. These environments were big enough that a PL/SQL application framework was a necessity, not a nice-to-have. After having to design and write a complete PL/SQL application framework twice, from scratch, only to kiss it goodbye when I left the client due to contractual agreements, I decided it was time to write my own that could travel with me, akin to a doctor's "black bag" or mechanic's "toolbox."

This I did with a telecom employer in Colorado where it had a long and trouble-free life. With the next employer, an energy consulting and software company in Houston, they already had some libraries in place. I was able to borrow pieces of the Starter framework to fill the few holes they had. Finally at the non-profit for which I currently work, Starter has been used on a number of new projects that would have otherwise spent weeks developing certain features from scratch. For my current employer, and so the framework could finally be open-sourced, I removed many of the more complex and industry-specific libraries it had accumulated over the years, paring it down to just the essentials, the services and features most PL/SQL development can use. I also updated it to make it easier to use and compatible with 10g. Starter was open-sourced early 2008. Since then only a handful of bugs have been found, and they all pertained to the new sections added in late 2007, testifying to its robustness.

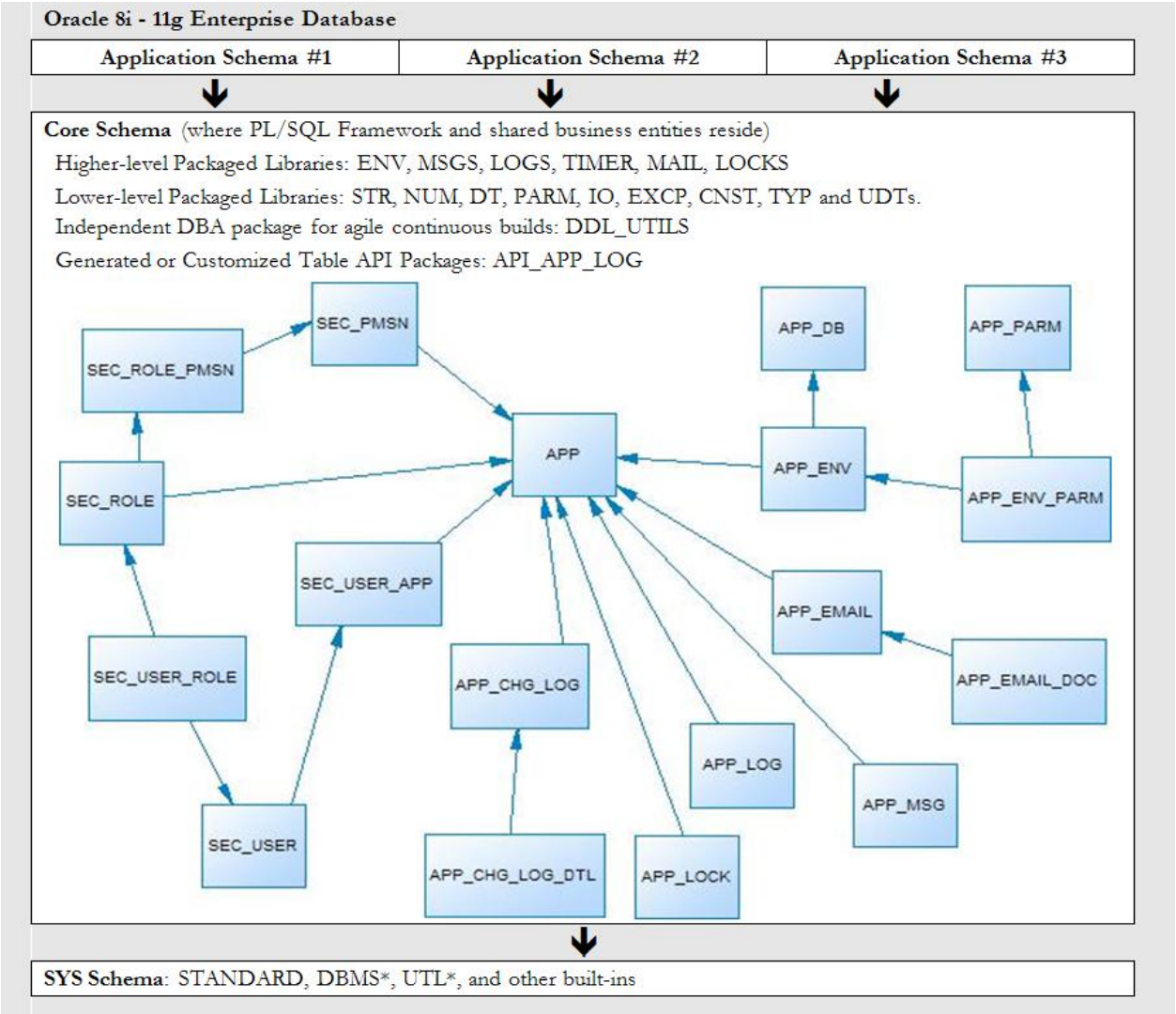
In preparation for IOUG Collaborate in 2010, Starter was again revisited and slightly updated to work with 11g. See the ReleaseNotes_2.0 file in the Docs folder for a full description of the recent changes.

The Big Picture

The framework is meant to be installed in a single schema, and then shared by the applications running inside multiple schemas on a given database. As a shared set of services, I like to install the framework in a schema named CORE or COMMON, but you may use any name desired. It can also be installed in an existing schema if you already have one that serves as the repository of re-usable PL/SQL code.

Examine the diagram below. The outer gray box represents the Oracle database. Each inner white box represents a schema in the database. The big, black arrows represent dependence on "lower-level" Oracle schemas, where the objects in the lower layers are granted to the higher layers, and accessed by public or private synonyms. The PL/SQL objects in the Core schema depend on items in SYS. And the application-owning schemas depend on the Core schema and on SYS.

The Core Schema box lists the PL/SQL libraries included in Starter, in dependency order from higher to lower layers. The simple conceptual model displays the main data structures used by Starter to provide the services listed in the table of contents on the first page



Multiple Applications on a Single Database

One of Starter's initial design goals was to provide a single installation of the framework on a database, all shared by any database applications running in their individual schemas on the same database. This is the default configuration and is accomplished by:

- Granting Starter objects to the application schema that needs the framework.
- Creating private synonyms in the application schema for the Starter objects.
- Configuring application and environment metadata in the APP, APP_DB, APP_ENV and APP_ENV_PARM tables (see Parameters/Configuration for more detail).

Multiple Application Environments on a Single Database

Another design goal was to save Oracle license fees by allowing multiple environments for an application to run on the same database. So the application-owning schemas in the diagram above don't have to be separate applications; they could also be separate environments for a single application.

For example, let's say there is an application dedicated to credit card fraud detection (CCF for short). Let's say the company requires development, continuous build, testing, user acceptance, staging and production environments. Production must reside on its own database host, of course. And staging has to look just like production, so it must be on its own host[1]. But the other environments for the CCF app could conceivably run on the same database. One could create multiple schemas on a single, well-apportioned database, naming them CCF_DEV, CCF_BLD, CCF_TEST and CCF_UAT, all pointed at the single installation of Starter.

This setup is accomplished by:

- Ensuring that no code in any tier hard-codes the schema name.
- DDL scripts use the built-in functions USER or SYS_CONTEXT('userenv','current_schema') instead of hard-coding the application schema name.
- Using distinct schema names for each environment on the same database.
- As before, configuring application and environment metadata in the APP, APP_DB, APP_ENV and APP_ENV_PARM tables (see Parameters/Configuration for more detail).

Multiple Applications in a Single Schema

At one employer, the legacy system had sadly been architected with multiple applications all using the same shared schema. This creates problems too numerous to list here, but there was no time or budget to fix the situation. So Starter was refactored a bit to accommodate this edge case. Starter is configured to accept this setup by:

- Configuring entries for the various applications in the APP, APP_DB, APP_ENV and APP_ENV_PARM tables as before. However, in this case, the applications that reside in the same schema will all have the same APP_ENV.owner_account value.

[1] Staging would be a good environment on which to do performance and load testing, since resources would not be shared by competing applications on the same database

- Backend processes, like DBMS jobs, begin life within the shared schema, instead of entering through the access account (as do front-end sessions). Because of this, there is no way for the framework to automatically determine which application that process belongs to. So the backend processes have to help the framework out a little bit by "identifying" themselves, telling the framework which application they belong to by calling ENV.init_client_ctx() or ENV.set_app_cd() at the top of the session.

Database Access Accounts

Since customers, users, application servers, developers and other databases all require access to your database to query data, execute packages and run an application, they have to be provided an account and a password to do so. It is not wise to give these clients the credentials of the schema that owns all the application objects.

Instead, it is best to create separate accounts for each of these database clients, for the purpose of entering the database. I like to call these "gateway" accounts or "access" accounts. They own almost no objects of their own, and have very few privileges.

In order to see and use the objects in the application schema, access accounts will either have their own set of private synonyms or will make use of an AFTER LOGON trigger which changes their default namespace[2] to the application schema, eliminating the need to manage many sets of private synonyms.

In particular, I'd like to speak of access accounts used by application servers. Most contemporary web applications are served up by code deployed to an application server. The application server accesses the application's database objects through its own access account. For performance reasons, the application server pre-establishes a number of cached physical connections, all connected to the database through the same access account. This is called a pool of connections, or connection pool. Unfortunately connection pools hide the actual end user on the front end. The database is only aware of the access account's connection. An application server's access account is configured in the APP_ENV.access_account field of each environment.

For instrumentation, security, and auditing features of the framework, as well as VPD, RLS, FGA, DBMS_MONITOR and standard AUDIT features in Oracle, it is critical to know the identity of the end user. This is known as end-to-end user identification.

End-to-End Identification

The identity of the database client can be passed to the database in a number of ways. Only two of these ways are really practical and worth discussing.

The first method involves calling a little PL/SQL procedure upon every database request, then another as that request is closed to clean out the identity metadata so that the next user assigned to that cached connection won't be accidentally identified as the last user. This is proven and works for Oracle v7/v8 or

[2] This is accomplished by a single EXECUTE IMMEDIATE statement in the AFTER LOGON trigger which does an "ALTER SESSION SET CURRENT_SCHEMA = <application schema;"

higher. Since the extra trips to the database are through a cached physical connection, the overhead of even thousands of database requests is negligible.

Assuming users have to log into your application, the application should have access to their login credentials. Their login ID can be passed to the database. To do this with Starter:

- Modify your application server's database connection classes/code to include the following PL/SQL call upon every database request, passing the ID they used to authenticate:

```
env.init_client_ctx(:UserID);
```

- When done with that trip to the database and relinquishing the connection back to the pool, again modify the connection class/code to make the following call:

```
env.reset_client_ctx();
```

The second method used to be straightforward, calling the OracleConnection.setClientIdentifier() method. But this method was deprecated with 9iR2, and has been replaced by the new OracleConnection.setEndToEndMetrics method instead. This method ensures the identifier you place into the metric piggy-backs on every connection and is available to the backend processes.

<Add working example here>

Proxies, Disabled Roles, VPD and Other Security Oddities

As a product that has passed rigorous tests so it can be used by the most sensitive of government agencies, Oracle has a dizzying array of security features that can be combined and configured in countless ways. VPD, globally authenticated and enterprise users, label security, disabled roles, global application contexts, and proxy accounts are a just a handful of these features. I have no idea how Starter would behave when these features are used or combined. As an independent open-source author providing this software to the community free of charge with no expectation of anything in return, I simply don't have time to test the framework for all possibly compatibility.

If you find flaws in the framework or installation scripts due to nuances in your environment, please create bugs in SourceForge against it, or even better, become a contributor to the project and check in the fixes needed to get it working for you.

Giving Back

Some heavy users of Starter may find a bug or two. Others will build upon Starter creating their own library to standardize the use of some Oracle built-in, or to standardize a vertical industry process, etc. If the enhancement isn't proprietary, consider giving back to the open source community by becoming a member of sourceforge and the plsqlframestart project, submitting your improvements to the community for review and inclusion.

Installation

When you've done it once, it doesn't seem so daunting. I routinely install the entire framework in under 30 seconds. But the first time often feels a bit tentative. Hopefully the following steps will clarify what is going on under the covers so there is less ambiguity and so the installation goes smoothly.

Obtain the Software

1. Download the zip file and documentation from SourceForge (1 min.)
2. Unzip to the directory of your choice. I used C:\Dev\Core. (10 seconds)

Pre-installation Decisions

Before you run the install script, there are a few decisions to make.

- Will this framework be installed in an existing schema or would you like a new schema to be created for you?
- Will the framework's segments be allocated within an existing tablespace or a new one? Would it be better to have another tablespace for the framework's index segments?
- Would it be helpful to see sample configuration data in the framework's application, database, environment and parameter tables/views?

Pre-installation Preparation: New Schema

Installing the framework to a new schema is the installer's default behavior. The `__InstallStarterFmwk.sql` script prompts you for the schema name where you would like the framework objects installed. Just enter the name of the new account you would like created. Script `_create_role_and_user.sql` will be called to create the two framework-related roles and the new account to hold the framework objects. With your DBA hat on, examine `_create_role_and_user.sql` to ensure it meets departmental or company-wide policies.

Pre-installation Preparation: Existing Schema

Some shops already have a "base", "core" or "common" account into which they have compiled re-usable libraries. A little more work is required to use an existing account like this.

1. Open `__InstallStarterFmwk.sql` and comment out line 85. No need to call `_create_schema.sql`. However, line 86, `_create_roles.sql` must still execute as it creates two roles that should be granted to applications using the framework.
2. The existing account should have the following privileges before installation:
    a. The basics (like CREATE SESSION, CREATE TABLE, etc.)
    b. CREATE ANY SYNONYM and DROP ANY SYNONYM
    c. CREATE ANY CONTEXT
    d. ALTER SESSION
    e. SELECT on v_$instance, v_$mystat, v_$session and v_$version (only SYS can grant)

Pre-installation Preparation: New and Existing Tablespaces

If you want the framework's objects in their own tablespaces, create one for tables and one for indexes.

Provide the names of the tablespaces when prompted for default, index and temporary tablespace names.

Pre-installation Preparation: Oracle Directories

The framework provides features that allow writing of general files, log files and email files to the database host's file system. The install script will prompt for the path to these directories. You may point all three directories to the same path, or use multiple OS directories. The script defaults to C:\temp, C:\temp\logs and C:\temp\mail. Your paths can certainly be Unix/Linux paths if this is being installed to a non-Windows server. But for the file I/O features to work, the paths should exist and have the appropriate read/write privileges granted (to the OS user used for the installation of the Oracle database server).

When you feed the install script the paths to the directories on your DB host, the script will create three Oracle directory objects pointing to those paths.

Pre-installation Preparation: Remove Sample Apps and Metadata

If you wish to see the example data configured in the APP, APP_DB, APP_ENV, APP_PARM and APP_ENV_PARM tables, do nothing. The install script will create sample application accounts and populate the framework tables accordingly.

However, if there is no need to see sample apps or sample data population:

1. Open `__InstallStarterFmwk.sql` and comment out lines 127-147 and 240-256. These lines created example accounts. Save it.
2. Open `_populate_sample_data.sql` and comment out the lines that insert rows for the TKT or BLG applications, leaving only the rows for the base CORE framework itself.

Install

Once the initial research and homework is complete, installation is simple.

1. Log in through SQL*Plus as SYS or an account with privs to act as SYSDBA.
2. Run `__InstallStarterFmwk.sql` and follow the prompts (20 seconds).
3. If something goes badly, use Ctrl+C to quit the script and exit SQL*Plus. To clean up and try again, log in again as a SYSDBA and execute `__UninstallStarterFmwk.sql`.

Post-installation

All that remains is to create private synonyms to framework objects inside the application schema(s) that will make use of the framework's features, as well as populating the application, database, environment and parameter tables for that application.

1. To create the new set of private synonyms, log in as the framework account and run `_create_synonyms_for_fmwk_objects.sql`. When prompted for "fmwk_consumer", enter the account name which owns PL/SQL objects that will make use of the framework.
2. Adding the application metadata to the framework tables is a little more involved. Please see the Parameters/Configuration section for that task.

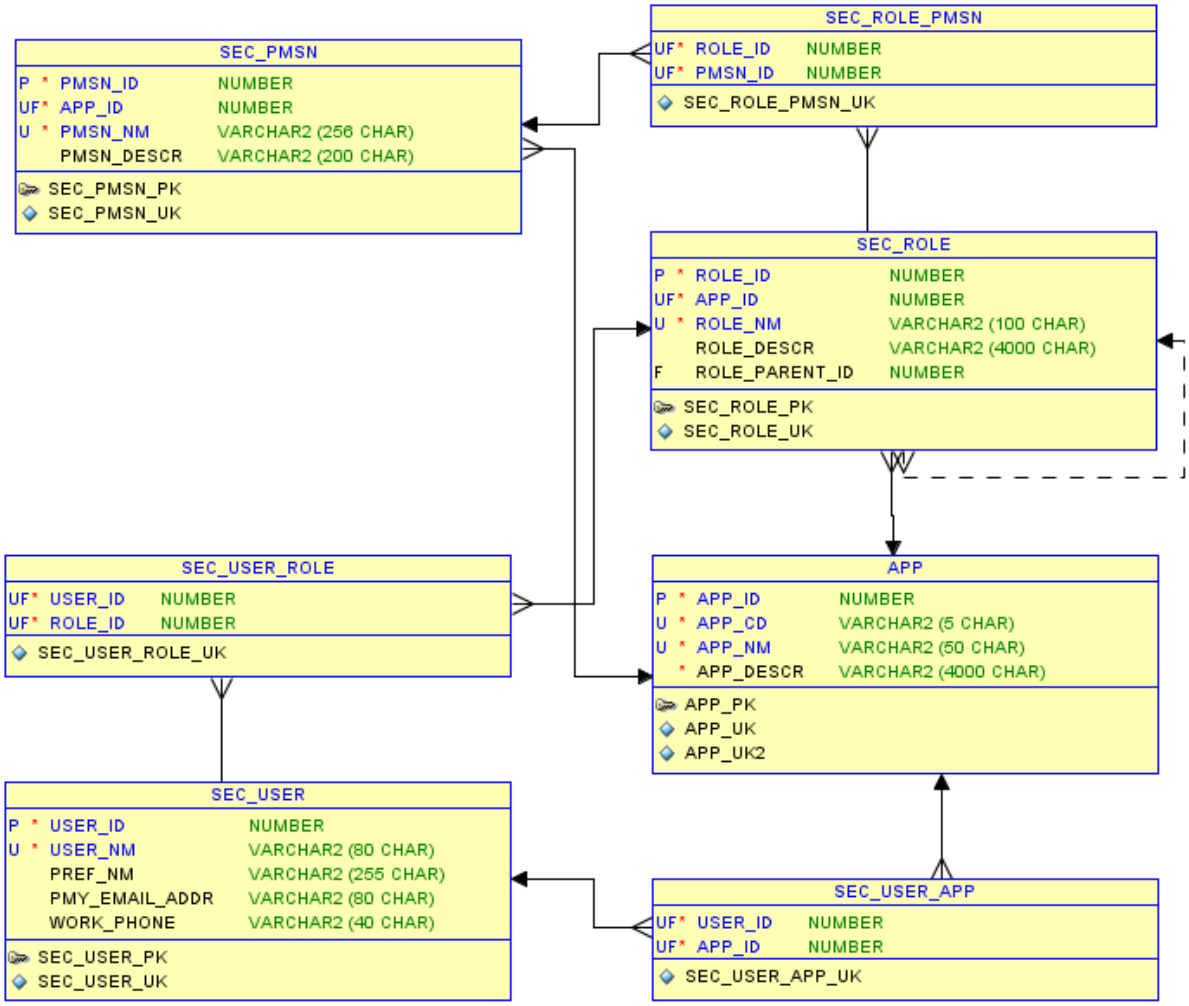
Security

The security portion of Starter is the most straightforward feature. It is also the most independent of the features. If all your application needed was the security piece, with a few tweaks, the rest of the framework could be thrown away, leaving the security tables and the views.

The security structures were designed to implement a slightly modified version of the standard RBAC security model (Role Based Access Control). The deviations from RBAC are that Starter uses the term "User" instead of "Subject", and simplifies by removing the Organization, Session and Operation entities from the model. Feel free to tweak the Starter data model if you need these pieces of the RBAC model added back in.

What remains of RBAC are Users, Roles, Permissions and the mappings between them. These are embodied in the SEC_USER, SEC_ROLE, SEC_PMSN, SEC_USER_APP, SEC_USER_ROLE, and SEC_ROLE_PMSN tables. With the self-referential foreign key on SEC_ROLE, the RBAC concept of hierarchical roles is preserved as well.

Security Physical Model



As you may have noticed from the model above, users do not "belong" to an application. It is anticipated that within the corporate walls where multiple applications run on a single database, that a given user is probably authorized to use several of those applications. This mapping happens in SEC_USER_APP. Roles and Permissions, however, are application-specific, as seen in the app_id column on both SEC_ROLE and SEC_PMSN. Once those are defined, it is simply a matter of mapping the permissions to the roles where that permission is allowed, and mapping the users to the roles they are allowed.

In order to begin benefitting from the security feature, do the following:

1. There must already be a row in the APP table for your application. See Parameters/Configuration for more information on this step.

```
INSERT INTO app (app_id, app_cd, app_nm, app_descr)
VALUES (app_seq.nextval, 'INV', 'Invoicing', 'Customer Billing Application');
```

2. Insert rows into SEC_ROLE for roles that your application requires (Administrator, Manager, Clerk, etc.). With the presence of the app_id column, this table can contain the roles for many applications.

```
INSERT INTO sec_role (role_id, app_id, role_nm, role_descr)
VALUES
   (sec_role_seq.nextval,
    (SELECT app_id FROM app WHERE app_cd = 'INV'),
    'Administrator',
    'Company administrator and CFO');

INSERT INTO sec_role (role_id, app_id, role_nm, role_descr)
VALUES
   (sec_role_seq.nextval,
    (SELECT app_id FROM app WHERE app_cd = 'INV'),
   'Customer Service',
   'Serves customers calling with billing questions.');
```

3. Insert rows into SEC_USER for the users who are authorized to use your application. Map this authorization in SEC_USER_APP.

```
INSERT INTO sec_user (user_id, user_nm, pref_nm)
VALUES (sec_user_seq.nextval, 'johndoe', 'John Doe');

INSERT INTO sec_user_app (user_id, app_id)
VALUES ((SELECT user_id FROM sec_user WHERE user_nm = 'johndoe'),
        (SELECT app_id FROM app WHERE app_cd = 'INV'));
```

4. With the users entered, map them to the roles they are allowed in SEC_USER_ROLE.

```
INSERT INTO sec_user_role (user_id, role_id)
VALUES ((SELECT user_id FROM sec_user WHERE user_nm = 'johndoe'),
        (SELECT role_id FROM sec_role WHERE app_id =
                (SELECT app_id FROM app WHERE app_cd = 'INV')
            AND role_nm = 'Customer Service'));
```

5. Now the hard part, determining permissions, storing and mapping permissions in SEC_PMSN and SEC_ROLE_PMSN. Read on…

Permissions

The simplicity of the security model is due to the anticipated use of the permissions table. Most home-grown security information models over-engineer in an attempt to anticipate and accommodate every single thing to which a user or role might be authorized. But the modern RBAC model eliminates the clutter of typical novice security modeling by transferring any real complexity to the permissions entity.

The permission, kept in SEC_PMSN.pmsn_nm, can be as large-grained or fine-grained as you wish. The permission can protect a business process, a report, a screen, a widget on the screen, the actions a user can take on said widget, etc. It is easy to handle such a diverse range of granularity by inventing and applying a naming scheme to your permissions. For example, you might set a permission naming syntax like this:

{functional area}.[page].[tab/menu item].[control].[permitted action]

The optional [permitted action] can be applied at any level in the permission scheme. For example, if you had an "edit" permission, it could be applied to the page, the field, the completed form, etc. Here is an actual permission from a system I'm currently working on:

"resource.AssignmentMeetingSchedule.editAssign"

The front end developers will work their magic to parse these permissions to enable access to pages and controls, disable widgets, dynamically construct page components, etc.

Once the permissions are built and entered into SEC_PMSN, it is only a matter of mapping the permissions to the roles to which they apply, and your security data is now complete.

Dynamic Security Views

The Security model comes with a few views named SEC_PMSN_VW, SEC_ROLE_PMSN_VW, SEC_ROLE_VW, SEC_USER_APP_VW, and SEC_USER_ROLE_VW. These views are "self-adjusting" and should be granted to any application schemas that will use them. They self-adjust in that when the application reads from these views, they automatically determine which application is performing the query and only return the security data that pertain to that application.

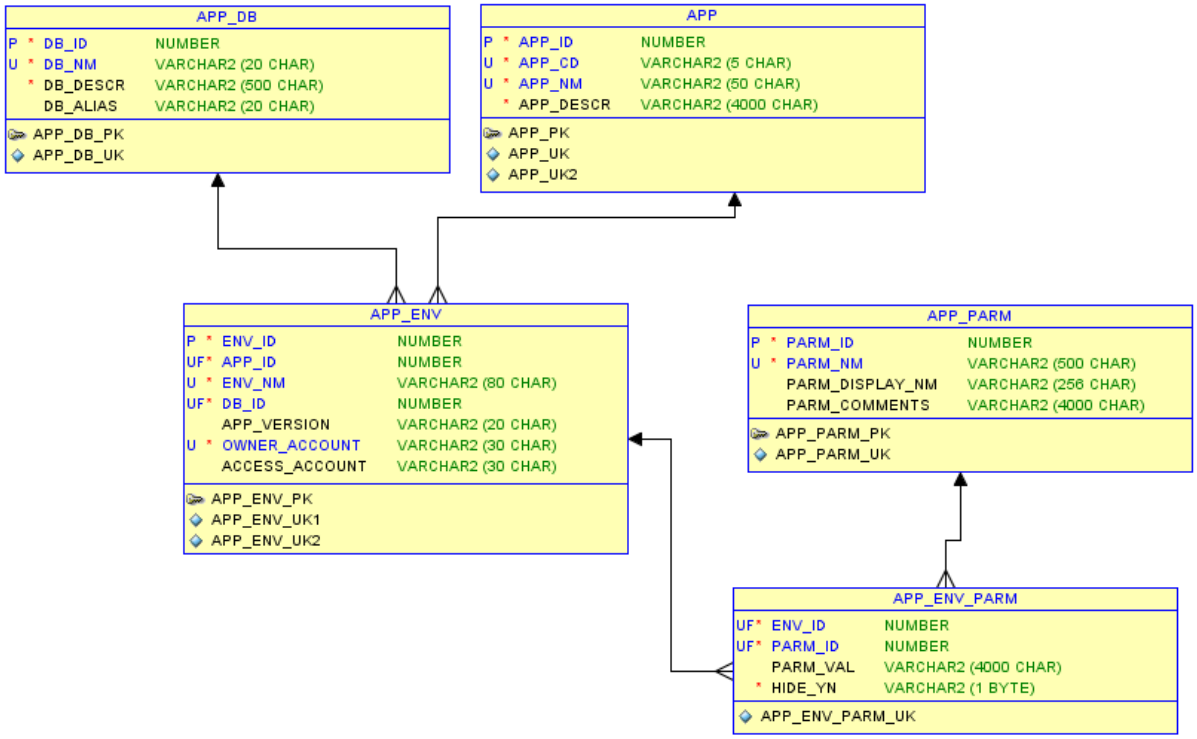
Unlike the Parameters, Starter's security was built with the idea that roles and permissions would be the same in every environment. So the need to tie into APP_ENV for environment-specific security settings is not called for. However, the list of users does typically change per environment. For example, the SEC_USER table will often contain a number of fake accounts in the Dev and Testing environments for use by unit testing and QA testing. To handle environment-specific users, one can write a script that uses MERGE to ensure certain accounts exist if the script is being run on certain databases. Cruise Control can be used to run this script for you after each build.

Parameters/Configuration

Like security, most modern applications require parameters and properties that can be dynamically altered at runtime without having to recompile code. This generally means storing them in an XML file, LDAP directory server, or database. Since I'm a database bigot and this is an Oracle framework, my vote is to store them in the database.

Parameters and properties are things like SMTP host name/address, remote FTP password, LDAP over SSL port, host address and directory for incoming ETL files, user inactivity timeout period and so forth. I refer to these values collectively as application parameters.

Environment-Specific Application Parameters Physical Model



Parameters can be used under the covers by the business or data layer, displayed for review or editing in the front end, or hidden and inactivated entirely. Display is usually accomplished in a listbox or drop-down, showing the values in the parm_nm column, or the optional parm_display_nm column. Hiding a parameter is accomplished by setting the APP_ENV_PARM.hide_yn flag column to Y in APP_ENV_PARM, which we'll talk about shortly.

Unlike security, it is typical for parameter values to be environment-specific. For example, your company probably has an FTP server for testing and another for production, an SMTP relay for testing and one for production, and three PeopleSoft databases for dev, testing and production. For this reason, the APP_PARM table does not have a column for the parameter value. Instead, values are kept in APP_ENV_PARM, where each parameter has a value for every environment configured in APP_ENV for your application.

As you can imagine, with each application having multiple parameters, and each parameter having multiple values, the normalized APP_ENV_PARM table can be difficult to read on its own. So Starter includes some more user-friendly, "self-adjusting" views named APP_ENV_PARM_VW, APP_ENV_VW, and APP_PARM_VW. These views join to the parent tables for you, to include the more readable name and description columns. Like the security views, these views should be granted to the applications participating in use of the framework. When an account with access queries these views, they automatically determine which application is making the request and present only those environments and parameters that pertain to that application for that database[3].

For these dynamic views to work properly, the application, database and environment metadata need to be set up first.

Note: Most examples in the following sub-sections were taken from `_populate_sample_data.sql`*, found in Starter's main install folder. Turn to that script to see these statements in context, in order, and within the configuration steps as a whole.*

Configuring the APP Table

As seen in the Security section it is exceedingly simple to add a row to APP for each application that will participate with the framework.

```
INSERT INTO app (app_id, app_cd, app_nm, app_descr)
VALUES (app_seq.nextval, 'INV', 'Invoicing', 'Customer Billing Application');
```

Configuring the APP_DB Table

Next add a row in APP_DB for each database that is part of your development, testing and operations environment. The function ENV.get_db_name() is critical to the functioning of the framework. It uses `SYS_CONTEXT('userenv', 'db_name')` to determine the database on which it is operating. So the APP_DB.db_nm value must match that of the database's SID or SERVICE_NAME.

```
INSERT INTO app_db (db_id, db_nm, db_descr, db_alias)
VALUES (app_db_seq.nextval, 'X911', 'Dev and Test','NRETST');

INSERT INTO app_db (DB_ID, DB_NM, DB_DESCR, db_alias)
VALUES (app_db_seq.nextval, 'X912', 'Prod', 'NREPRD');
```

Configuring the APP_ENV Table

Now a few rows are added for each application environment. Assuming an application will have a development, continuous build, testing, staging and production database environment, APP_ENV will require five rows for that. Here are three of those environments, two of them configured to co-exist on

[3] Caveat: PL/SQL Triggers and Views are unable to use invoker rights. Due to this, the session's application code, stored in an application context and an in-memory structure managed by the ENV package MUST be set prior to the view first being queried. To ensure this happens, I've found the best workaround to be an AFTER LOGON trigger in the access and application owner accounts which call ENV.set_app_cd(), that is until Oracle adds invoker rights to views.

the same database host (repeat APP_ENV population for each application that runs on the same database and shares the Starter framework schema):

```
INSERT INTO app_env (env_id, app_id, env_nm, db_id, app_version, owner_account,
access_account)
VALUES ( app_env_seq.nextval
       ,(SELECT app_id FROM app WHERE app_cd = 'INV')
       ,'Billing Dev'
       ,(SELECT db_id FROM app_db WHERE db_descr = 'Dev and Test')
       ,'2.0'
       ,'BLG'
       ,'BLG_FD'); -- FD stands for "front door" access account
INSERT INTO app_env (env_id, app_id, env_nm, db_id, app_version, owner_account,
access_account)
VALUES ( app_env_seq.nextval
       ,(SELECT app_id FROM app WHERE app_cd = 'INV')
       ,'Billing Test'
       ,(SELECT db_id FROM app_db WHERE db_descr = 'Dev and Test')
       ,'2.0'
       ,'BLG_TST'
       ,'BLG_TST_FD');
INSERT INTO app_env (env_id, app_id, env_nm, db_id, app_version, owner_account,
access_account)
VALUES ( app_env_seq.nextval
        ,(SELECT app_id FROM app WHERE app_cd = 'INV')
        ,'Billing Prod'
        ,(SELECT db_id FROM app_db WHERE db_descr = 'Prod')
       ,'1.0'
        ,'BLG'
        ,'BLG_FD');
```

Configuring the APP_PARM Table

A parameter begins configuration with a new row added to APP_PARM, as in this example:

```
INSERT INTO app_parm
  (parm_id, parm_nm, parm_display_nm, parm_comments)
VALUES
  (app_parm_seq.nextval, 'Timeout', 'Request Timeout',
   'How long to allow requests to churn before returning the Timeout error.');
```

Configuring the APP_ENV_PARM Table

Once the environments are defined and the required parameters are named, the two are brought together in APP_ENV_PARM in order to set the environment-specific parameter values. Here are two example inserts showing the parameter value receiving a slight adjustment for a different environment:

```
INSERT INTO app_env_parm (env_id, parm_id, parm_val)
VALUES ((SELECT env_id FROM app_env ae JOIN app a ON a.app_id = ae.app_id WHERE
a.app_cd = 'INV' AND ae.env_nm = 'Billing Dev')
       ,(SELECT parm_id FROM app_parm WHERE parm_nm = 'Default Log Targets')
       ,'Screen=Y,Table=Y,File=Y');

INSERT INTO app_env_parm (env_id, parm_id, parm_val)
```

```
VALUES ((SELECT env_id FROM app_env ae JOIN app a ON a.app_id = ae.app_id WHERE
a.app_cd = 'INV' AND ae.env_nm = 'Billing Test')
       ,(SELECT parm_id FROM app_parm WHERE parm_nm = 'Default Log Targets')
       ,'Screen=N,Table=Y,File=N');
```

Once APP_ENV_PARM is fully populated, the dynamic views go to work. Logged into an application account, issuing a query on the updateable view like this:

```
SELECT t.*, t.rowid FROM app_env_parm_vw t;
```

allows one to view and alter only the parameter values that pertain to the current environment.

Auditing

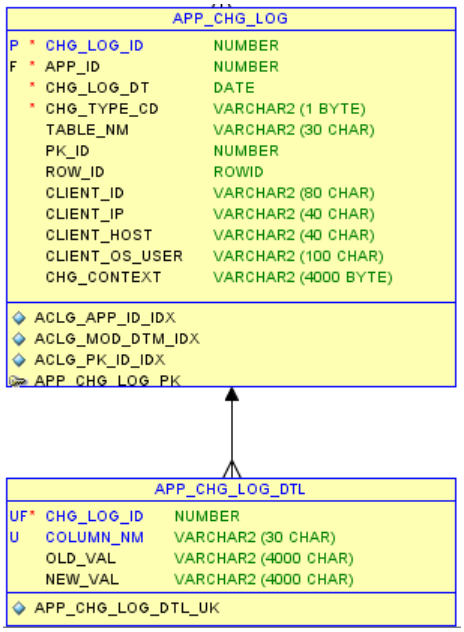
Like Security, Starter's fine-grained DML auditing feature is straightforward. It only involves two tables, the ENV package, and a trigger generator.

There are at least five or six different ways to record changes to data over time. Starter takes a generic approach, with one table, APP_CHG_LOG recording the metadata about the row being changed. The child table, APP_CHG_LOG_DTL keeps one row of old/new values for each column changed in a row.

Once a table is identified as containing columns that should be audited, in particular financial and legal data where it is very important who did what and when, it is very easy to start auditing them. Starter comes with a script named `gen_audit_triggers.sql`. As the application schema, run this script. It will output an audit trigger to the screen for each table[4] in the schema. Modify the resulting triggers, commenting out or deleting the code tracking changes to columns that are of no interest. Then compile the triggers. Fine-grained changes will now be tracked in the Application Change Log tables.

The best part about this common framework-driven approach to auditing is the synergy with the ENV library. The audit triggers call upon the ENV functions to transparently record exactly which user made the change, when, what app the triggered the audit, which IP and host they connected from, etc.

The generic approach to auditing changed columns is fine for what I call "emergency auditing." This sort of audit data is only queried in emergencies, in the rare case the company is sued, or someone misappropriates funds, or a regulatory agency questions recent rate hikes, etc. The way the changes are stored "vertically", in the attributive APP_CHG_LOG_DTL table, makes it somewhat slow and difficult to retrieve and/or flatten the data into a single, more easily-readable row of changes. Due to this performance drawback, if your application has requirements to frequently view changes, say for a history page or on-demand report, consider a materialized view on top of Starter's audit tables, or create a custom audit table and trigger to populate it. For a typical custom audit table, a copy of the base table is created, replacing each base column with a pair of columns to record the old/new values.



Partitioning the Audit Tables

If you anticipate auditing large volumes of data, and you have license to Oracle Enterprise Edition with Partitioning, consider partitioning the two tables. This makes it trivial to drop months or years off the back end of the table as they age out of the corporate audit data retention period.

[4] If auditing is needed only for a few tables, modify the WHERE clause of that script's driving cursor.

The additional DDL to do so is commented out in `_create_base_tables.sql`. It is written to 11g and takes advantage of reference partitioning for APP_CHG_LOG_DTL. If still using 10g, add chg_log_dt to APP_CHG_LOG_DTL and include that column in the partitioning key.

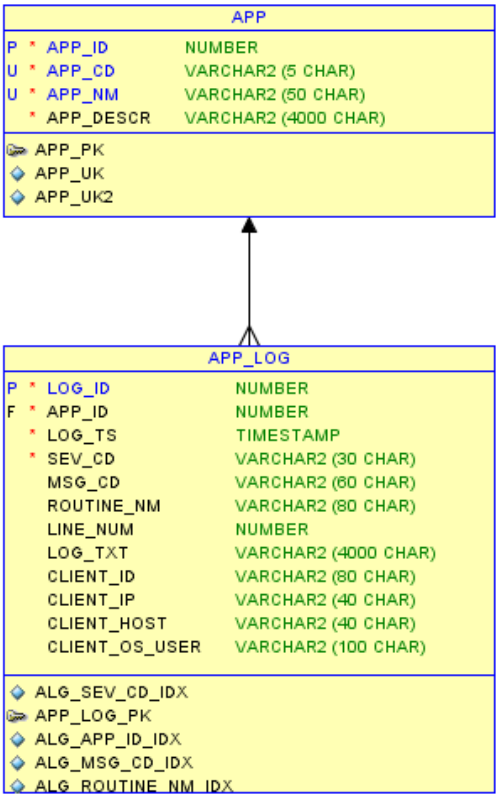
Instrumentation (Debugging, Monitoring, Timing & Logging)

A hallmark of mature development is well commented and fully instrumented code. Like testing, it can be seen as time consuming and a chore, but the payback is enormous when things go wrong, as they sometimes do. Instrumented code is easy to measure, tune and troubleshoot. When called upon, it can provide all the information needed to easily diagnose performance, security or functional problems.

The Starter framework provides simple interfaces for application, error, warning, informational and debug logging, as well as routines to easily take timing measurements and make the session more monitorable, visible and auditable.

Logging

Logging, the ability to write information about the application's state to some output medium, is essential to every system with any kind of backend processing.



Starter provides the ability to log to the file system, a table, or the screen (the DBMS_OUTPUT buffer). If more advanced forms of output are needed, like http, udp, ftp, DBMS or OS pipes, etc., either enhance Starter, or use something like log4plsql. I find the ability to log to the screen useful in development, and the ability to log to a generic logging table useful for all other environments.

The library for logging is the LOGS package. Despite a personal standard against the plural form of entity and object names, the singular "LOG" could not be used as it was an Oracle keyword. The LOGS package is the highest level package in Starter, in that it depends on all but two of the other Starter packages to function. So, all of Starter needs to be installed to make use of the LOGS library.

Although the logging table model to the right seems rather busy, the only thing you really need worry about is what type of a log message it is, and the message itself. The rest of the fields can be filled automatically by the LOGS library of routines, assuming things are configured correctly.

Logging Destinations

Starter comes with a small set of required parameters to function. One of these, named *Default Log Targets*, is set to "Screen=Y,Table=Y,File=Y" in development, and "Screen=N,Table=Y,File=N" in all other environments. Changing the flags from Y to N and vice versa toggles the output destination of any logging the application's PL/SQL code is performing. These changes can be done dynamically, at runtime (even in production) simply by altering the parameter values in APP_ENV_PARM_VW.

If unit testing in development, it may be helpful to override the parameter's setting for the current test condition. The parameter value in the table could be changed, but that's a manual step. To automation this change, the log targets can be overridden using LOGS.set_log_targets().

Almost half of the routines in the LOGS package are dedicated to viewing or altering the destination directory and file name of any logging output going to a file on the database host. See the package specification for further details.

Errors

Serious processing errors, raised as PL/SQL exceptions, are those that should probably halt processing right away. It is a best practice to ban the use of WHEN OTHERS in PL/SQL exception clauses, trapping only expected errors, allowing unexpected errors to immediately bubble to the top of the call stack, rollback the transaction and report where the error occurred. And yet almost every Oracle shop I've visited or worked in seems to have a love affair with WHEN OTHERS, capturing every exception, logging it, and then either accidentally or purposefully hiding the error, or re-raising it as they should.

If your code has some known exceptions it needs to trap and log, capture the known errors by exception name[5]. Do any necessary cleanup (like explicitly closing cursors). Then log the error and the context around it using LOGS.err(), which will automatically re-raise the error after writing to APP_LOG.

See the package spec for full details, but briefly there are two versions err(). LOGS.err() with no arguments is the easiest to use, but allows no additional processing context to be logged along with the error. It has three formal parameters with defaults. The only one that really makes sense to change is the first one, the i_reraise parameter. If you change the default to FALSE, you can log the error and continue processing. If left TRUE, the exception will be re-raised after being logged. The second version of LOGS.err() allows additional context (like current loop iteration variable values) to be passed in the i_msg parameter, and will be kept with the error information logged.

It is essential that your shop set, adopt and enforce the use of a rigorous standard for handling expected exceptions. The unconvinced reader is directed to Steven Feuerstein's online writings on this subject for the reasons behind this statement.

The LOGS library is a basic, but easy-to-use framework for handling exceptions. It is based on principles taught by Feuerstein in his *Oracle PL/SQL Best Practices* book, chapter 5: Exception Handling. The reader is **highly** encourage to read and re-read that chapter until the principles become second nature. In short:

- Ban the use of WHEN OTHERS except when hiding or averting the error is intentional.
- Handle only expected exceptions.
- When exceptions are handled, use a standard interface[6] that can record much of the context of the error for you, then re-raise the error.

[5] Use PRAGMA EXCEPTION_INIT to bind the local exception to the expected Oracle error number.
[6] In Starter that interface is the LOGS package. The EXCP package also provides an API for raising an exception explicitly, but EXCP.throw should not be called directly.

Warnings

Warnings are expected exceptions that may or may not warrant further attention. These are the sorts of conditions that should probably be reviewed in the next few hours to ensure it isn't a symptom of something worse, but in general, warning messages can wait. Error messages cannot.

To log a warning message, call LOGS.warn(). It carries the i_msg formal parameter as well, so additional context can be included along with the warning message.

Informational Messages

Finally, there is application logging. This kind of logging is often used to form an historical data trail indicating how long some process too to run, which files were available at the time of process initialization, which organization is no longer found in the master data database, etc. Anything you want permanently and immediately recorded about the session can use LOGS.info() to get a message stored into APP_LOG.

Monitoring

Oracle comes with the DBMS_APPLICATION_INFO interface to place identifying program information into the session metadata found in V$SESSION. These are module, action and client_info.

> *"Registering the application allows system administrators and performance tuning specialists to track performance by module. System administrators can also use this information to track resource use by module. When an application registers with the database, its name and actions are recorded in the* `V$SESSION` *and* `V$SQLAREA` *views."* (Oracle® Database PL/SQL Packages and Types Reference, 10*g* Release 2)

What the 10g overview does not say, is that these values also show up in active session history and in the USERENV namespace of SYS_CONTEXT. They are a wonderful way of providing real-time, dynamic views into what your program is actually working on. This is particularly useful when the bug seems to indicate that the program is stuck or hanging. Looking at these values can tell you what it last started doing, provided the code is instrumented to identify the latest module or action before starting on the next step.

Starter provides a simple interface to place these values into the session, and to remove them, named ENV.tag() and untag(). You should find these easier to type and remember than the DBMS_APPLICATION_INFO routines and their nuances.

```
env.tag('REPORTS.print_and_send_ps', 'Open cur_read_ps_db cursor');
logs.dbg('Reading and storing all problem/solution rows');
FOR l_rec IN ps_dml.cur_read_ps_db LOOP
   ...
END LOOP;
env.untag();
```

Debugging

Bugs happen. There is no such thing as perfect code. There is always one more bug. The sooner one internalizes these maxims, the sooner fantasy can be traded for wisdom and prudence, and defensive coding can begin. Most PL/SQL shops wish they had a way to easily debug production code, but the task seems daunting and the effort is usually abandoned before it begins. Some leave DBMS_OUTPUT.put_line (poor man's debugging) in their production release code. But resources are expended sending these messages to nowhere since production code doesn't read from or display the DBMS_OUTPUT buffer. If it isn't left in production, it is often added when the problems arise, at the expense of users and the business as downtime must be taken to recompile the highly coupled packages, invalidating session state and ruining the reputation of applications that must have high availability. More advanced Oracle development teams do have a form of debug logging in place, often with debug levels to control how many of the messages are recorded and how deeply. But this code is usually left in place, constantly expending resources and expanding size on disk or in tables.

Given these restrictions and problems, the ideal debugging framework would:

- Allow debug messages to remain in production code, but consume little to no resources when debugging isn't "turned on."
- Allow the debugging to be dynamically turned on for new sessions and existing sessions in Production. This means the switch ought to be table-driven.
- Allow debugging to be turned on only for one session, or one user, or one or more PL/SQL units.
- Provide immediate feedback, or visibility into debug messages that are now "going somewhere" with debugging turned on. This opposed to typical DBMS_OUTPUT calls that only show up when the session is over and the buffer is read.
- Automatically determine as much as it can about the end user, calling routine and connection metadata, so the developer does not have to worry about this.
- Be very easy to use, type and remember, otherwise the developers will discard it.

Starter provides all this, and does so in a very simple package. With the right configuration, this is how one calls the debug API:

```
logs.dbg('Processing salesperson '||l_rec.emp_name||'. Iteration '||l_idx);
```

It's that easy. I write my PL/SQL in pseudocode, with lots of comments indicating the intent of each section. Later when the code is implemented, I'll wrap those comments with logs.dbg() and add some context to make the messages more useful and traceable. This accomplishes two best-practices in one shot: instrumentation and good commenting. The underlying framework determines everything else for the developer, including the calling end-user, and the name of the calling packaged procedure/function.

It couldn't be easier to use the debugging logging procedure of the LOGS library.

However, there are a few more useful debug routines in LOGS, and three parameters that must be set up for each application environment. The useful routines are those that can get and set (override) the file IO parameters, and the one that can change the default debug toggle, turning debugging on for the

session. The later is named LOGS.set_dbg() and is overloaded to accept both Boolean and string toggles. The former are named: get_targets, get_log_dir, get_log_nm, get_log_path, set_targets, set_log_dir, set_log_nm and set_log_path. It is unlikely these will ever come in handy for most shops.

The three parameters are: Debug, Debug Toggle Check Interval and Default Log Targets. The last one was mentioned earlier. Remember it tells the LOGS library where messages passed to LOGS routines should be output to. Debug Toggle Check Interval is a simple integer value, telling the LOGS library how often it should check the Debug parameter in the APP_ENV_PARM table to see if it has changed since the last time it checked. The default is one minute. This parameter is the "magic" that takes all the LOGS.dbg() calls peppered in production code to just start working, making it appear as if debugging was injected into existing sessions.

The final Debug parameter is the one that enables the remaining features of Starter's debug framework, allowing Debugging to be dynamically turned on in any environment by simply changing the value of a table column. You can use APP_ENV_PARM_VW, or a query like the following to view and edit the Debug parameter values:

```
SELECT ep.env_id
      ,e.env_nm
      ,ep.parm_id
      ,p.parm_nm
      ,ep.parm_val
      ,ep.rowid
  FROM app_env_parm ep
  JOIN app_env e
    ON e.env_id = ep.env_id
  JOIN app_parm p
    ON p.parm_id = ep.parm_id
  JOIN app_db d
    ON d.db_id = e.db_id
  JOIN app a
    ON a.app_id = e.app_id
 WHERE d.db_nm = UPPER('&db_name')
   AND p.parm_nm LIKE '%Debug%'
   AND a.app_cd = '&app_cd';
```

The Debug parameter value is flexible, varied and case-insensitive. To turn on debugging for the entire schema, just change the value to all, Y, on, yes, or TRUE. To turn it back off, change it to none, N, off, no or FALSE. If you wish to turn on debugging only for a particular end user, change the value to user=<client ID>. If debugging an existing session, and session SID is known (see V$SESSION), change the value to session=<session ID>. If you want the debug messages spilling into APP_LOG to be limited to a particular package, packages or other PL/SQL units like triggers or procedures, change the value of Debug to unit=<pkg1,proc1,proc2,trigger1,etc.>

Here is a chunk of example code showing the use of timing, application logging, error logging, monitoring and dynamic debug logging:

```
CREATE OR REPLACE PACKAGE BODY reports
AS
```

```
PROCEDURE print_and_send_ps
(
  i_email_addr IN VARCHAR2 DEFAULT NULL
)
IS
   l_lines    typ.tas_maxvc2;
   l_email    CLOB := EMPTY_CLOB();
   l_filename VARCHAR2(128) := 'rpt_probsol_'||TO_CHAR(SYSDATE,'YYYYMMDD')||'.txt';
   l_loop_idx INTEGER := 0;

   PROCEDURE handle_line(i_line IN VARCHAR2) IS
   BEGIN
      l_lines(l_lines.COUNT+1) := i_line;
      l_email := l_email || i_line || CHR(10);
   END handle_line;

BEGIN
   excp.assert((env.get_env_nm <> 'ProbSol Prod' AND i_email_addr IS NOT NULL)
               OR env.get_env_nm = 'ProbSol Prod',
               msgs.fill_msg('Missing Parameter', env.who_am_i, 'i_email_addr'),
               TRUE);

   timer.startme('read_db_write_file');

   logs.dbg('Checking for file '||l_filename);
   IF (io.file_exists(l_filename)) THEN
      logs.dbg('Deleting file '||l_filename);
      io.delete_file(l_filename);
   END IF;

   env.tag(i_module => 'REPORTS.print_and_send_ps', i_action => 'Open cur_read_ps_db
cursor', i_info => '');
   logs.dbg('Reading and storing all problem/solution rows');
   FOR l_rec IN ps_dml.cur_read_ps_db LOOP

      l_loop_idx := l_loop_idx + 1; -- placed to demo variable watches and conditional
loops

      IF (l_lines.COUNT = 0) THEN -- Add header if nothing in report yet
         handle_line(str.ctr(RPT_DIV_LINE));
         handle_line(str.ctr('Printout of the Problem/Solution Database'));
         handle_line(str.ctr(TO_CHAR(SYSDATE, 'YYYY Month DD')));
         handle_line(str.ctr(RPT_DIV_LINE)
                     ||CHR(10));
      END IF;
      handle_line('Type [' || l_rec.prob_src_nm || '] Key [' ||
                  l_rec.prob_key || '] Error [' || l_rec.prob_key_txt || ']');
      handle_line('Comments:');
      handle_line(CHR(9) || l_rec.prob_notes);
      handle_line('Solution #'||l_rec.seq||':');
      handle_line(CHR(9) || l_rec.sol_notes || CHR(10));
      handle_line('-----------------------------------------');

   END LOOP;
```

```
   env.untag();

   logs.dbg('Writing '||l_lines.COUNT||' lines to file '||l_filename);
   io.write_lines(i_msgs => l_lines, i_file_nm => l_filename);

   timer.stopme('read_db_write_file');
   logs.info('Reading DB and writing file took
'||timer.elapsed('read_db_write_file')||' seconds.');

   timer.startme('write_email');
   logs.dbg('Sending report to director if in Production, otherwise to given email
address');
   mail.send_mail(i_email_to => i_email_addr,
                  i_email_subject => l_filename,
                  i_email_body => l_email,
                  i_env_list => 'ProbSol Dev, ProbSol Test');
   mail.send_mail(i_email_to => 'bcoulam@boguscompany.com',
                  i_email_subject => l_filename,
                  i_email_body => l_email,
                  i_env_list => 'ProbSol Prod');
   timer.stopme('write_email');

   logs.info('Writing email took '||timer.elapsed('write_email')||' seconds.');

END print_and_send_ps;

END reports;
```

Timing

It is said that one cannot manage that which is not measured. If all of your system's database routines are sub-second, single-row operations, you might get by with have no timing instrumentation in place. However, anything longer or more complex than that should be timed. Taking timing measurements imposes virtually no overhead on the system, **so leave timings in place in production code**.

Starter's time measurement features are kept in the TIMER package. It is one of the only completely independent pieces of Starter and can be installed standalone. TIMER includes three simple routines:

- startme()
- stopme()
- elapsed()

Each of them takes a single parameter, which is the name of the thing being measured.

Each time startme() is called with a new name, a new timer is created. Naming each "timer" gives us the ability to have multiple timers all running at once. With this ability, each piece of a backend process can be timed: the overall driver or process, each routine called, their subroutines, loop iterations, etc. Timers can be sampled, or peeked into, by calling elapsed(). Once a timer has been stopped, the time returned by elapsed will be static. Timers can be restarted, although I've never found a reason to do so.

The elapsed times measured can be written statically (every time) using the logging routines, or dynamically (only when switched on) by embedding them in debug calls. Once timers are in place and recorded to table or file, the measurements can be monitored or mined to watch for system degradation or improvement, assist with application or database upgrade regression testing, etc.

If lazy or in a great hurry, the TIMER routines can be called without passing a timer name. If TIMER.startme is called without a name, a generic name will be generated for it. However, this defaulting means you are limited to just that one timer in that session. So always name your timers.

Here is an example:

```
timer.startme('outer loop');
FOR l_rec IN cur_load_waiting LOOP
   timer.startme('load');
   loader.process(l_rec);
   timer.stopme('load');
   logs.dbg('Processing load ID '||l_rec.id||' took '||timer.elapsed('load')||'
seconds.');
END LOOP;
timer.stopme('outer loop');
logs.dbg('Loading took '||timer.elapsed('outer loop')||' seconds.');
```

DDL Operations

Starter comes with an independent package named DDL_UTILS. It depends on nothing else within the framework and can be used standalone. This package contains a large set of functions that is primarily meant for both manual and automated build processes, where new and modified database objects are migrated from development to higher environments. You can use DDL_UTILS routines during the initial stages of populating a system's schema with objects, as well as during migration to production and during the system maintenance lifecycle.

Besides the usual benefits of being repeatable, robust, easy-to-remember and use, the value of DDL_UTILS routines is a) preventing errors that aren't really errors from halting your automated build processes, and b) transparently handling version-specific and partitioning nuances for certain DDL operations.

The one thing DDL_UTILS can't do well is wrap CREATE and ADD operations. So you won't find any helper routines for these operations. Here is a quick intro to the sorts of public services available as functions and procedures in DDL_UTILS. Where a routine takes an Oracle object name as a parameter, the parameter is not case-sensitive. See the package spec for full documentation.

Category	Service	Short Purpose
Retrieve Metadata	GET_CONS_COLUMNS	Returns a comma-delimited list of the columns that make up a constraint.
	GET_DEFAULT_VALUE	Returns a VARCHAR2 version of the LONG default value for the given column.
	GET_NUM_ROWS	Returns the number of rows currently in the given table.
	GET_SEARCH_CONDITION	Returns a VARCHAR2 version of the LONG check constraint condition.
	GET_TBL_BY_CONS	Find the table name for a given constraint.
	GET_TBL_BY_IDX	Find the table name for a given index.
	GET_VERSION	Return DDL_UTIL's current version number.
	GET_DB_VERSION	Returns the major version of the database as a number, even if on older versions w/o DBMS_DB_VERSION installed.
Drop Object	DROP_TBL	Drops the named table, with or without data. See package and routine notes for some fancy stuff it does with dependent foreign keys.
	DROP_PK	Drops the named primary key. Will also drop the underlying index, unless directed otherwise.
	DROP_UK	Drops the named unique constraint. Will also drop the underlying index, unless directed otherwise.
	DROP_FK	Drops the named foreign key.
	DROP_CHK	Drops the named check constraint.
	DROP_IDX	Drops the named index.
	DROP_SYN	Drops the named synonym.

	DROP_PUB_SYN	Drops the public synonym.
	DROP_OBJ	Drops an Oracle object not covered by the other DROP_ routines.
	DROP_COL	Drops a column from a table.
	DROP_COL_LIST	Drops a set of columns from a table.
	DROP_SEQ	Drops the named sequence.
	DROP_ALL_OBJ	Drops all objects inside the invoking schema. Can be fed an exclusion list. Does not drop DDL_UTILS by default.
	DROP_ALL_PRIV_SYN	Drop all private synonyms.
	DROP_ALL_PUB_SYN	Drop all public synonyms.
Modify Column	SET_UNUSED	Sets a column to unused.
	REMOVE_DEFAULT	"Removes" a column's default value.
	CHANGE_COL	Coordinates the steps behind changing the datatype or shortening a column's length.
	RENAME_COL	Renames a column.
Rename Object	RENAME_CONS	Renames a constraint.
	RENAME_IDX	Renames an index.
	RENAME_SEQ	Renames a sequence.
	RENAME_TBL	Renames a table.
Modify Sequence	MOD_SEQ_CACHE	Modifies a sequence's cache spec.
	RESET_SEQ	Reset's the sequence's NEXTVAL based on the current MAX value in the column populated by the sequence (useful after manual population of reference tables that will use a sequence for all future row insertions).
Modify Table	REMOVE_PARALLEL	Removes parallelism from a named table or index
	REMOVE_PARALLEL_ALL	Removes parallelism from all objects in the database.
	ENABLE_ROW_MOVEMENT_ALL	Finds partitions with row movement disabled and enables them.
	ADD_LOGGING_ALL	Ensures all permanent (heap, IOT and partitioned) tables and indexes have LOGGING set on.
Rebuild or Move	REBUILD_IDX	Rebuilds an index in its existing tablespace, unless a new tablespace is specified by the caller. Will automatically compute statistics as it is rebuilding unless explicitly told not to.
	REBUILD_UNUSABLE	Rebuilds any indexes that went unusable.
	MOVE_IDX	Moves an index from one tablespace to another.
	MOVE_TBL	Moves a table from one tablespace to another.
Existence	OBJ_EXISTS	Boolean. Determines if the named Oracle object exists.
	ATTR_EXISTS	Boolean. Determines whether a contained item, like a column, type attribute or type method, is found in the containing object.
	DATA_IS_FOUND	Boolean. Determines if a given table (or an optional partition) has data in it or not.

Gather Statistics	ANALYZE_INDEX	Analyzes a named index per a set standard use of DBMS_STATS.
	ANALYZE_SCHEMA	Analyzes a named schema per a set standard use of DBMS_STATS.
	ANALYZE_TABLE	Analyzes a named table per a set standard use of DBMS_STATS.
Transparent FK Management	RECREATE_DEP_FKS	Reads an in-memory structure to re-create the foreign keys lost upon deletion of a parent table or primary key.
	PRINT_DEP_FKS	Prints out the DDL to re-create all the foreign keys dependent on a given table or unique constraint.
	PROCESS_CONSTRAINT_LIST	Reads an array of constraint specifications and drops, recreates or adds each one, depending on action discerned.
Object Status and Access	RECOMPILE	Routine by Solomon Yakobson, updated by Feuerstein and myself to work from 8 to 11g. With grace, finds invalid objects and recompiles them in the correct dependency order.
	REFRESH_GRANTS	Grants full or read-only access on the invoking schema's objects to a named role or schema. An exclusion list can be passed.
Test DB	ECHO	Little utility used during framework testing to ensure DDL_UTILS is valid and past the package invalidated for the session error.

Database, Application and Connection Metadata

One of the most powerful and valuable libraries in Starter has been referred to several times in sections above. It is the ENV package. It provides around 40 functions to return attributes about the session, call stack, database, application and connected end user. These functions are used by numerous parts of the framework to simplify the interfaces and transparently provide valuable services that most framework writers typically force their developers to provide. For example, ENV contains the routines to automatically figure out the package and routine name of the calling routine. This allows framework code like the LOGS routines (msg, err, warn, info, dbg) to transparently determine the caller, so the developer doesn't have to pass it.

For most applications, the only direct use of ENV will be to send the authenticated end-user's credentials to the backend using ENV.init_client_ctx() in the application server's connection pool code, place processing markers into V$SESSION using ENV.tag/untag(), and maybe to put a bind variable value into Starter's default application context using ENV.set_ctx_val/clear_ctx_val.

However, it is still useful to know what is available in ENV, in case you ever have the need for a robust way of determining, for example, the database version, or the RAC instance name. Here is brief listing. See the ENV package specification for full details:

Service	Short Purpose
DOMAIN	Constant that contains your company's domain, e.g. mycompany.com.
GET_CLIENT_ID	Returns end-user identifier
GET_CLIENT_IP	Returns client IP address (typically that of the app server unless passed end-user IP is passed by frontend through env.init_client_ctx)
GET_CLIENT_HOST	Returns client host machine
GET_CLIENT_OS_USER	Returns OS account of client
GET_CLIENT_PROGRAM	Returns program from V$SESSION
GET_CLIENT_MODULE	Returns module from V$SESSION
GET_CLIENT_ACTION	Returns action from V$SESSION
GET_SESSION_USER	Returns name of the connected schema.
GET_CURRENT_SCHEMA	Returns name of the current schema (can be diff from connected schema if altered).
GET_DB_VERSION	Returns whole integer for the database version number.
GET_DB_NAME	Returns the SID/SERVICE name of the database.
GET_DB_INSTANCE_NAME	Returns the name of RAC instance currently connected to.
GET_DB_INSTANCE_ID	Returns the ID of the RAC instance currently connected to.
GET_SERVER_HOST	Returns the hostname of the database host.
GET_SID	Returns the session ID (short name)
GET_SESSION_ID	Returns the session ID (longer, clearer name)
GET_OS_PID	Returns the OS process ID (useful for performance

	investigation and session killing from *nix OS)
GET_SCHEMA_EMAIL_ADDRESS	Returns default fake email address to use in From field of emails from within the database.
GET_DB_ID	Uses the db_name returned by sys_context to query APP_DB and return a numeric ID for the current database.
GET_APP_ID	If the app_cd is given, returns the app_id from the APP table. If the app_cd is missing, will determine the app_id dynamically and transparently.
GET_APP_CD	If the app_id is given, returns the app code from the APP table, otherwise will determine app_cd from get_app_id().
GET_ENV_NM	Queries APP_ENV to determine the name of the current environment, given the current schema, database name and application.
GET_DIR_PATH	Queries all_directories view to find the directory path behind a given 9i-style directory name.
VLD_PATH_FORMAT	Ensures that there is a directory slash character after a given path. If the given path does not end in a slash, one will be appended.
WHO_CALLED_ME	By default, returns the name of the package or standalone one level further up in the call stack, which represents the caller of the routine that called this function.
WHO_AM_I	By default, returns the name of the package or standalone that called this function.
GET_ROUTINE_NM	Given a package name and line number, returns the name of the routine within which that line number currently falls.
LINE_NUM_HERE	Looks in the call stack to the given depth and returns the line number from which line_num_here was called.
CALLER_META	Returns all the metadata about the caller at the given level in the call stack. It is anticipated that the only consumer of this proc will be the LOGS library routines (msg, err, warn, info and dbg).
TAG_SESSION/TAG	Sets MODULE, ACTION and CLIENT_INFO in v$session to the provided values. Use this routine frequently to instrument your code and DDL/DML upgrade scripts.
UNTAG_SESSION/UNTAG	Sets MODULE, ACTION and CLIENT_INFO in v$session to NULL.
SET_CTX_VAL	You associate this routine with an application-specific context during application context creation. Then call this routine when setting the

	values of attributes within the context.
SET_APP_CD	Takes a short code for a given application (found listed in the APP table), and sets it as the value for the "app_cd" attribute in the Core application context.
INIT_CLIENT_CTX	Takes whatever info the frontend wants to pass about the end user and places it in application context and the USERENV namespace.
RESET_CLIENT_CTX	Must be called by the front-end layer controlling transactions and access to the connection pool. This empties the client context and resets package state, so that the next user who inherits these in-memory objects doesn't also inherit the same values.
CLEAR_CTX_VAL	This routine will set the value of the named attribute within the given namespace to NULL.
CLEAR_CTX	This routine will set the value of all attributes within the given namespace to NULL.

Assertions

There are numerous books and articles -- not just from PL/SQL gurus like Feuerstein -- but programming gurus that speak of coding defensively, and design and code by contract. One of the tenets of coding by contract is always checking assumptions. Most of us code to "sunny day" scenarios where nothing goes wrong. But it is the sign of a mature programmer who programmatically ensures that everything she assumes to be true at the top of a routine actually is true, before proceeding. This double-checks that the "contract", the verbal or written agreements between the caller and the called routine is being upheld, and yields highly robust code.

Starter's EXCP package offers a flexible routine, named assert() to check these assumptions. Its default usage is very simple, but the optional parameters render it flexible for other scenarios. In most situations the violated condition should halt further program processing. In rare cases though, you may wish the processing to continue after logging the violation. Here are a few examples:

Notes:

- Anything that can be turned into a Boolean expression can serve as the first parameter to assert().
- Condition violation/solution messages will be printed to the screen and to the logging table.
- Being one of the lowest-level packages, the EXCP package could not make circular references to the LOGS library. Errors and messages from EXCP.throw and EXCP.assert automatically go to the screen (where they are wasted if the caller has no visual interface, like a DBMS job), and they are also logged to APP_LOG through the APP_LOG_API.

Numbers:

```
excp.assert(i_run_id > 0, 'Run ID must be a positive integer.');
excp.assert(LENGTH(l_str) < 4000,'Message is too long');
```

Dates:

```
excp.assert(i_start_dtm >= dt.get_sysdt,
      'Time travel violation. Start date must be in the future');
```

Strings:

```
excp.assert(l_code IN ('U','X','Z'), 'Valid codes are U, X and Z.');
```

Boolean:

```
excp.assert(l_continue_flg = TRUE, 'Continue Flag is false.');
```

NULL conditions:

```
excp.assert(l_stmt IS NOT NULL, 'Provided statement can''t be empty.');
excp.assert(l_var IS NULL, 'Variable l_var already had a value.');
```

Optional named exception handling:

```
excp.assert(l_state_busy, NULL, TRUE, 'excp.gx_row_locked');
```

Optional log and continue:

```
excp.assert(i_expr => l_var IS NULL,
       i_msg => 'Variable l_var already had a value.'
       i_raise_excp => FALSE);
```

Constants, Types, Cached Reference Data

Most applications use a few enterprise-wide, sometimes industry-wide, constants or “magic numbers” that should be stored in a central location. The code or value is then re-usable. If the value ever changes, it only need be changed in one place, instead of the hundreds of pieces of code that reference it.

Developers often get a little lazy and hard-code the literal value in their front, middle or data layer code, adding unnecessary redundancy, moving parts, and stress points. But they should be pointing at a constant kept in the database, either behind a public constant, a public function, or a parameter from a table. If there is a good likelihood of the value changing, putting the value behind a named parameter is the best approach. But if no one can foresee the value changing in the near future, it will perform better to assign the value to a named constant in a package specification, or to load the values from a reference table into an in-memory PL/SQL structure (best for backend processes that can easily access that structure within the session).

Constants

In its earliest version (1997) Starter had one package, named “C”, to hold all the constants for all the applications using the framework. This turned out to be a very bad idea. Constants were constantly (excuse the pun) being added to C. Every time that happened, all packages for all the applications at the telecom company where this was first applied would go invalid. Quite annoying.

Today the package of constants is named CNST. It only comes with a handful of very generic “universal” constants. Less generic constants should go in the package specification of the library to which they most closely align. If you have a constant that is true for your entire enterprise and is never going to change, put it in CNST. Otherwise, add it to the specification of the PL/SQL package to which it applies.

Here is the complete list of included constants:

```
-- Lengths
PAGEWIDTH    CONSTANT INTEGER := 80;
MAX_COL_LEN CONSTANT INTEGER := 4000;
MAX_VC2_LEN CONSTANT INTEGER := 32767;

-- Basic return codes, following the Unix convention of non-zero indicating failure
SUCCESS CONSTANT PLS_INTEGER := 0;
FAILURE CONSTANT PLS_INTEGER := 1;

-- Basic numeric represenation of boolean values, following the conventions found
-- in C and other languages, where true=1 and false=0.
TRUE CONSTANT PLS_INTEGER := 1;
FALSE CONSTANT PLS_INTEGER := 0;

-- Basic values for flag columns. These are preferred over 1's and 0's because
-- their meaning is instantly clear. If 1's and 0's are needed for SQL against
-- a table with numeric flags, use DECODE to map Y/N to 1/0.
YES CONSTANT VARCHAR2(1) := 'Y'; -- used for *_flg columns
NO CONSTANT VARCHAR2(1) := 'N'; -- used for *_flg columns

-- Message/log severity codes
ERROR CONSTANT VARCHAR2(10) := 'ERROR';
WARN  CONSTANT VARCHAR2(10) := 'WARN';
INFO  CONSTANT VARCHAR2(10) := 'INFO';
```

```
AUDIT CONSTANT VARCHAR2(10) := 'AUDIT';
DEBUG  CONSTANT VARCHAR2(10) := 'DEBUG';

-- System-wide symbols, strings and tokens
SEPCHAR CONSTANT VARCHAR2(2) := ': ';
PIPECHAR CONSTANT VARCHAR2(1) := '|';
DELIMITER CONSTANT VARCHAR2(1) := ',';
DIR_SEPCHAR CONSTANT VARCHAR2(1) := '/'; -- change to "\" for Windows OS

-- Substitution character. Strings that requires substitution/replacement at
-- runtime will be wrapped in this character. This is used mainly by the MSGS
-- library in its operations upon standard messages with placeholders in APP_MSG.
SUBCHAR CONSTANT VARCHAR2(1) := '@';

-- Other generic strings
UNKNOWN CONSTANT VARCHAR2(1) := 'U';
UNKNOWN_USER CONSTANT VARCHAR2(10) := 'UNKNOWN';
UNKNOWN_STR CONSTANT VARCHAR2(10) := 'Unknown';
```

Types

Starter also includes a very generic and useful set of types, subtypes and empty collections in the package TYP, many of them used by the framework. They are:

```
-- Subtypes
SUBTYPE t_maxobjnm IS VARCHAR2(30); -- maxlength for Oracle object name
SUBTYPE t_maxfqnm  IS VARCHAR2(61); -- fqnm = fully qualified name
SUBTYPE t_maxcol IS VARCHAR2(4000);
SUBTYPE t_maxvc2 IS VARCHAR2(32767);
SUBTYPE t_msg    IS t_maxcol;
SUBTYPE t_rc     IS PLS_INTEGER;

SUBTYPE t_mime_type IS VARCHAR2(100); -- used by MAIL and UTILS

-- PL/SQL associative arrays
TYPE tab        IS TABLE OF BOOLEAN        INDEX BY BINARY_INTEGER;
TYPE tad        IS TABLE OF DATE           INDEX BY BINARY_INTEGER;
TYPE tan        IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;
TYPE tas_small  IS TABLE OF VARCHAR2(30)   INDEX BY BINARY_INTEGER;
TYPE tas_medium IS TABLE OF VARCHAR2(255)  INDEX BY BINARY_INTEGER;
TYPE tas_large  IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
TYPE tas_maxcol IS TABLE OF t_maxcol       INDEX BY BINARY_INTEGER;
TYPE tas_maxvc2 IS TABLE OF t_maxvc2       INDEX BY BINARY_INTEGER;

-- Used as default values for formal parameters, and as assignment
-- sources for quickly emptying out a populated collection (can also
-- be accomplished by calling <collection>.DELETE() )
gab        tab;
gad        tad;
gan        tan;
gas_small  tas_small;
gas_medium tas_medium;
gas_large  tas_large;
gas_maxcol tas_maxcol;
gas_maxvc2 tas_maxvc2;
```

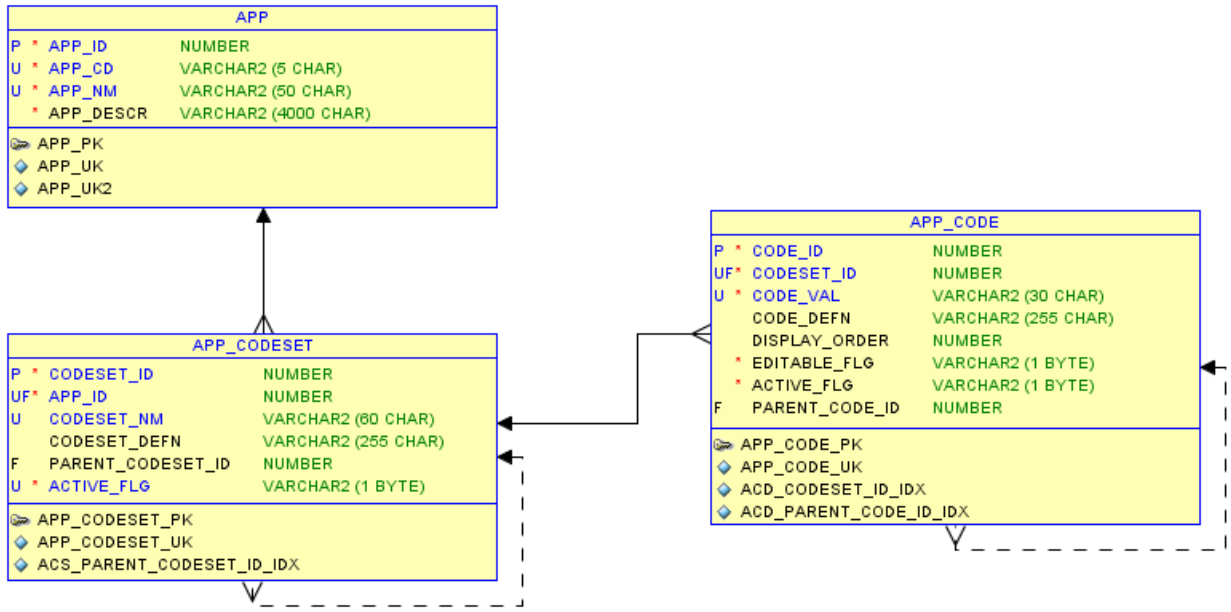
Reference Data and Cached Reference Data

Some development groups believe it is a good idea to place all “type”, “lookup” or “reference” codes and values in a single lookup table. I even had a Java architect convince me to go this route once, to save on the number of classes he had to build. In the end though, this approach causes numerous problems.

The largest of these are hobbling data integrity and screwing up the CBO from being able to create intelligent paths (see a great proof of this in Cost-Based Oracle Fundamentals by Johnathan Lewis).

The best practice for reference data is to create a separate lookup table for each set of distinct codes/values.

However, if yours is a shop where the single-table approach is demanded, Starter can accommodate. Embedded in `_create_base_tables.sql` is the DDL for the APP_CODESET and APP_CODE tables. Uncomment the DDL and run it. It will build:



As you can see from the model above, a codeset is a way of grouping codes. A codeset is owned by a single application. Codesets can be subsets of other codesets. Each code belongs to one codeset. And each code can have a parent code. The hierarchies established for both codesets and codes is more useful than they seem. For example, in menu/navigation panes of webpages, or in drop-down lists where the codes need to be grouped or visually separated. Codesets and codes can be inactivated by setting the active_flg to N. Codes can be ordered by sequentially numbering a codeset's codes using the display_order column. The editable_flg column can be used by screens that allow administrators to view/maintain the code lists, so that critical code values used by application code are not changeable.

When these tables are created, there are also self-adjusting views, sequence triggers, auditing triggers and a CODES library (PL/SQL package) that you may compile to gain additional built-in functionality. They scripts that you'll need to run are:

```
app_code_aiud.trg, app_code_bi.trg, app_code_vw.sql
app_codeset_aiud.trg, app_codeset_bi.trg, app_codeset_vw.sql
codes.pks, codes.pkb
```

Once compiled, be sure to grant them to the schemas using the framework. Once you have your lookup values inserted into the tables, you may begin to create child tables in application schemas that foreign key back to the APP_CODE table, using the CODE_ID column as the referable primary key. Unfortunately, there is no way to prevent the child table from using a code that is outside the codeset of interest. This is one of the drawbacks of using a single lookup table and should be clearly understood before picking this approach to name/value reference pairs.

String Manipulation

Starter comes with a handful of basic routines to make certain string operations simpler. Your mileage may vary and may require far more than this basic set provides. Build upon it as needed.

Service	Short Purpose
String Constants: NUL, TAB, CR, LF, LFCR, CRLF, FF, SP, DEL	Small collection of constants that remove the need to remember the ASCII code for the character, making string formatting code more readable.
CONTAINS_NUM	Will return a 1 if the given string contains a number, 0 if no number is found.
CTR	Will return the given str centered within the page width. If no page width is provided, the default set in CNST will be assumed. More useful for ASCII reports spooled from SQL*Plus.
EWC	“Equal-width columns”. Use this function to pad or trim a string down to the desired column width. CAUTION: This will NOT work if the output isn't viewed in a fixed-width font, like Courier. More useful for ASCII reports spooled from SQL*Plus.
FOREIGN_TO_ASCII	Reads a string and converts all "foreign" characters to plain ASCII characters.
FORMAT_TO_WIDTH	This function takes in a string and a width to "break" the string into lines of manageable width by means of inserted linefeeds. Paragraphs will be preserved. Other linefeeds and carriage returns will be removed.
GET_DIACRITIC_LIST	Returns long list of diacritical characters that can be transliterated to one of the basic ASCII printing characters.
GET_DIACRITIC_MAP	Returns the list of basic ASCII printing characters which is usually applied to the string returned by get_diacritic_list by the TRANSLATE function to produce a 1:1 mapping between the two.
GET_TOKEN	Takes a delimited string, uses parse_list to turn it into an array of tokens, then returns the desired token to the caller.
MAKE_LIST	Creates a delimited list of strings out of an array (nested table) of strings. Delimiter can be overridden.
NONASCII_TO_ASCII	Anything outside the range of the lower ASCII set (128-255) as well as Unicode characters, will be converted to Unicode escape codes, e.g. the Euro symbol CHR(128) will be returned

	as \20AC.
PARSE_LIST	Creates an array (nested table) of strings out of a delimited list of strings. Delimiter can be overridden and can be multi-character. NULL entries in the list can be used or ignored.
PURGE_STR	This function takes in a string and cleans out ALL non-printing characters except the SPACE character.
SPLIT_STR	This routine shares the same purpose as parse_list. However, it is implemented differently. It is less robust, but does operate at twice the speed as parse_list. If you have a need for a speedier parser, and your lists are not sparse, but well-packed and delimited, you might want to use split_str instead.
TRIM_STR	This function takes in a string and cleans off ALL non-printing characters from the left and right side.

Number Manipulation

Starter comes with a handful of basic routines to make certain numerical operations simpler. Your mileage may vary and may require far more than this basic set provides. Build upon it as needed. Also comes with a couple constants which define useful numeric format masks for TO_CHAR and TO_NUMBER operations.

Service	Short Purpose
GET_SET_DIFF	Compares two nested tables of number (independent type created by the Starter install), returning numbers that appear in one not found in the other. The usefulness of this feature is now moot with 10g's SUBMULTISET comparison operator.
IAN	Numeric. Returns 1 if the string is a number. Used in SQL.
IANB	Boolean. Returns true if the string is a number. Used in PL/SQL.
IS_EVEN	Numeric. Returns 1 if the number is even. Used in SQL.
IS_EVENB	Boolean. Returns true if the number is even. Used in PL/SQL.
IS_ODD	Numeric. Returns 1 if the number is odd. Used in SQL.
IS_ODDB	Boolean. Returns true if the number is odd. Used in PL/SQL.
MAKE_LIST	Creates a delimited list of numbers out of an array (nested table) of numbers. Delimiter can be overridden.
PARSE_LIST	Creates an array (nested table) of numbers out of a delimited list of numbers. Delimiter can be overridden and can be multi-character. NULL entries in the list can be used or ignored.

Date Handling

Starter comes with a handful of basic routines to make certain date operations simpler and to centralize a set of useful date-related constants.

Constant Group	Constant List
Constants for time periods used in date arithmetic. Particularly useful in specifying DBMS job intervals.	SECONDS_PER_DAY, SECONDS_PER_HOUR, SECONDS_PER_MIN MINUTES_PER_DAY, HOURS_PER_DAY DAYS_PER_WEEK MONTHS_PER_YEAR, DAYS_PER_YEAR ONE_SECOND, ONE_MINUTE, ONE_HOUR FIVE_MINUTES, TEN_MINUTES, FIFTEEN_MINUTES, THIRTY_MINUTES
Several useful date format strings for use in TO_CHAR and TO_DATE conversions. The Y2K masks use RR as the year (mostly moot now that the year 2000 has passed).	BATCH_DT_MASK Y2K_BATCH_DT_MASK Y2K_DT_MASK DT_MASK DTM_MASK SORTABLE_DTM_MASK TM_MASK
For use in epoch-related functions	UNIX_EPOCH

Functions offered:

Service	Short Purpose
DT_TO_EPOCH	Converts an Oracle DATE value to the Unix "seconds from epoch" value, with the epoch defined as Jan 01, 1970 (as the constant UNIX_EPOCH).
EPOCH_TO_DT	Converts a Unix "seconds from epoch" numeric value to an Oracle DATE.
GET_DAY_NAME	Given the day of the week (as an integer), returns the name of the day of the week.
GET_HIGHDT	Returns the common future date value, often used to fill NOT NULL “end date” columns. The high date serves as a default value.
GET_SYSDT	Returns the current date (with the time portion removed).
GET_SYSDTM	Returns the current date and time.
GET_SYSTS	Returns the current date and time as a TIMESTAMP return value.
GET_TIME_DIFF	Returns the number of time units between two dates. The units valid for expressing the numeric return value are: day, hour, minute, second, week, year
GET_TIME_DIFF_STR	Returns the period of time between two dates suffixed with a given unit of measure. The unit suffixes valid for expressing the string return value are: second, minute, hour, day, week, year, dhm, hm

MINUTES_TO_DHM	Given a measure in minutes, returns a string formatted in dd:hh:mm format, e.g. 6:2:03 means 6 days, 2 hours and 3 minutes (leading zeroes removed)
MINUTES_TO_HM	Given a measure in minutes, returns a string formatted in hh:mm format, e.g. 0:34 means 0 hours and 34 minutes (leading zeroes removed)
TRUNC_DT	Returns

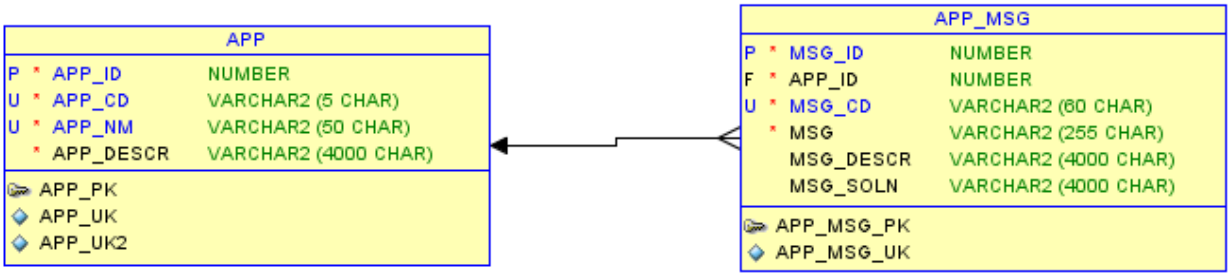
The reason for the get_sysdt, get_sysdtm and get_systs functions is to insert a layer between the application and the SYSDATE/SYSTIMESTAMP built-ins. If the application is rigorous about using these functions instead of the built-ins, when QA must test the application's behavior at a certain date in the future, instead of fiddling with the host OS internal clock, you can just modify where the DT routines pull their date from. Using this framework with a large system back in 1999 was very advantageous. In order to test how things would work come midnight entering the year 2000, I just had to change a couple functions.

Standard Messages

Over time most applications grow organically (a little weedy and out of control). With the coming and going of certain employees, styles of coding ebb and flow. Standards are a great way to solve this problem of changing whims. One standard that is often not even considered is a standard for the message strings that will be displayed to the user, typically in informational or error dialog windows.

Rather than allow each developer to invent their own flavor of a commonly-themed message to the end user, it is better to have a central place for re-usable messages that can be easily browsed, modified and added to.

Starter provides a simple structure so that each application participating in the framework can store its own set of standardized messages.



The msg_cd column is used to store a short name or code for the message. Try not to refer to the message by its msg_id, as it is a surrogate primary key and should have nor take on any intrinsic business meaning.

If the message is about detected errors for which there is a path forward, steps to take, or resolution, use the msg_soln (message solution) column to store that information.

Unlike the parameter structures, the message table has yet to join the “self-adjusting” scheme. That is planned for a future release, so that the unique key is app_id + msg_cd. Currently, only one of a given msg_cd value is allowed in the table, and querying app_msg from a given application schema will show all messages, not just the messages for that application.

Starter includes a simple interface to get a standard message string from the APP_MSG repository. It is MSGS.get_msg(). It is overloaded to allow lookup by either message code or ID. It is preferred to look the message up by code, stored in APP_MSG.MSG_CD. Here is an example of this function being used within an exception block in the LOCKS package:

```
        excp.throw(msgs.get_msg_id('Logical Lock Held'),
                   'Lock on '||i_lock_nm||' already held by '||lr_lock.locker_id||
                   ' within the '||lr_lock.app_id||' application. Try again later.');
```

If required, there are two routines to get at the msg_id using a known msg_cd (ENV.get_msg_id), or to find the msg_cd using a known msg_id (ENV.get_msg_cd). Although the author thought it a good idea to make these two routines public, he's never needed to use them in practice except privately inside the MSGS package body. Consider the user of these functions deprecated.

Certain messages aren't sufficient as static strings. In order to communicate something useful to the end user or system administrators, it requires context about the backend process when the error was detected. The MSGS package spec speaks of this in detail, but basically you create a standard message in APP_MSG that contains placeholder characters to surround the tokens that will be replaced dynamically when the message is called upon to log or display.

To replace the placeholders with dynamic contextual/state information, like variable values, call upon MSGS.fill_msg(). There are two versions. The first takes an array of strings. There is no limit to the number of values you can place in the array (in case your message has, say 23 placeholders). This version is quite flexible, but requires a little extra effort to declare, initialize and fill the collection variable. In most cases, the message will include five or fewer placeholders. So a simpler version with five string parameters was created to make it easier to replace placeholders with processing context.

Let's say you had a standard message named "FTP Login Failure". You define the msg text to be "Failure logging in to the @1@ FTP server using account @2@ and password @3@". In your application code that attempts to FTP something from the database host, in the exception section catching the FTP open failure, you could have a piece of code like this:

```
EXCEPTION
   WHEN lx_ftp_login_err THEN
      logs.dbg(msgs.fill_msg('FTP Login Failure', l_ftp_host, l_ftp_account,
                             l_ftp_pass));
```

Email from the Database

Sooner or later everyone with Oracle installed will encounter a need to send email from within the database. Typically the first reason will be some jobs that proactively monitor database, data and process health, emailing or sending SMS messages when something is off-kilter or really wrong.

There are numerous ways to accomplish this.

- The easiest is to write a file[7] to a directory on the DB host, and have an AT/cron job check the directory for new files, email them, then either delete or move them to an archive.
- Another method is to use a Java stored procedure inside Oracle to send an email to the SMTP server.
- One can also write emails to an email table, and have a middle tier daemon read the latest additions, send them, then update the table with the send status.
- The last two methods rely on built-in packages. Prior to 10g you could use UTL_SMTP. After 10g, UTL_MAIL is available, which is a better mail framework, built on top of the older UTL_SMTP.

Starter provides a simple interface, in the MAIL package, that lets you send an email from within the database using any of the first three methods above. The fourth method, using UTL_SMTP is also in the MAIL package but wholly commented out. The reason is due to problems, session death and performance issues using UTL_SMTP directly on 9i. If you require UTL_SMTP, you'll need to refactor the package a bit, and create some new unit tests before exposing the rewritten version to your developers.

If you are on 10g or higher, consider re-implementing the MAIL.send_mail() routine using UTL_MAIL instead of its default implementation (using a Java stored procedure).

The MAIL package requires a few parameters be set up in APP_ENV_PARM for each application schema that will use MAIL. They are Default Email Targets, Default Email File Directory, and SMTP Host. The SMTP host is obviously the name of the server that can handle your outgoing email. The directory is the name of the Oracle directory object pointing at the host directory where email files can be written. If you have no desire to ever write emails to the file system, remove the write_email routine and references to this parameter.

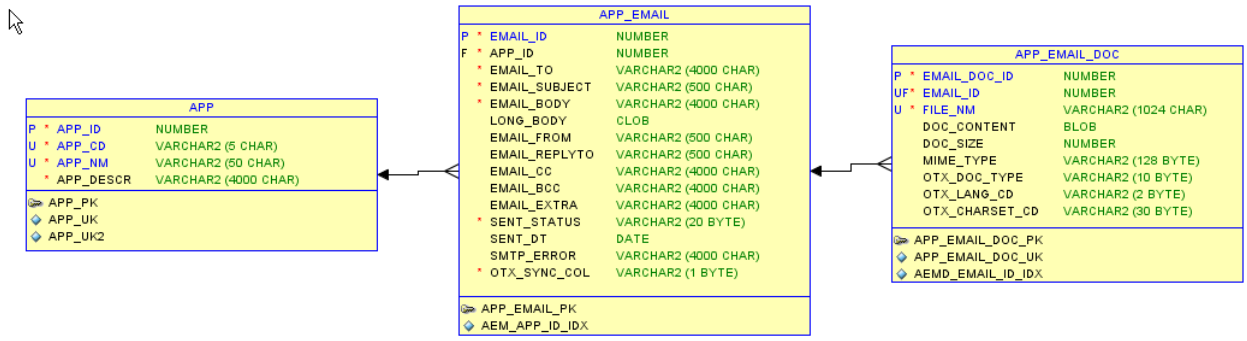
The Default Email Targets parameter is where "the magic" happens. Like the Default Log Targets parameter mentioned earlier, this defines three values and looks like this: SMTP=Y,Table=Y,File=N. The defaults setting assumes that most shops will not write emails to the DB host file system. It also assumes that most shops will want a queryable history of emails sent, so it stores emails to the APP_EMAIL table, as well as sending them out via SMTP. Alter these values for each environment per your application's needs. If there is ever the need to temporarily override the current settings, call MAIL.set_targets to do so.

When this initial setup is complete, it is a good idea to first get the SMTP connection working. For this reason, the is_smtp_server_avail() Boolean function was exposed in the package spec to enable easy

[7] Properly formatted per the email RFCs.

testing of this handshake with the SMTP server. Just call it in an anonymous block to test the completeness of your MAIL package configuration and network connectivity.

```
SET SERVEROUTPUT ON
DECLARE
BEGIN
   dbms_output.put_line('Call to is_smtp_server_avail returned ['||
                        util.bool_to_str(mail.is_smtp_server_avail)||
                       ']');
END;
/
```



Above is the data model of Starter's email structures. The ability to store the email to table is bound to work if Starter was installed correctly, and the Default Email Targets parameter includes the value "Table=Y". This can be tested by calling MAIL.write_mail(). However, most applications should not need to ever call write_mail or store_mail directly. Just call send_mail(). The values in the Default Email Targets parameter controls whether send_mail will route emails to table, file or SMTP.

So now that the connection is working, let's test the ability to send an email. See the package specification comments for full details. Suffice it to say that sending emails from within the database is very easy using send_mail(). In fact, there are only three required parameters: To, Subject and Body. Here is the simple example:

```
mail.send_mail('mymanager@mycompany.com','Daily Top SQL', i_report_txt);
```

You can pass in comma-delimited lists of email addresses in the To, Reply-To, CC and BCC input parameters. Valid email formats includes

- user_id@mycompany.com
- "Full Name" <user_id@mycompany.com>
- Full Name <user_id@mycompany.com>

You can flag an email as higher priority. You can ensure the email is not sent in certain environments. Here we send a report to a list of directors, copying a list of managers, with a high priority, asking for email processing only if this is a production or staging database.

```
mail.send_mail(i_email_to => i_director_list, i_email_subject => 'Monthly Downtime',
               i_email_body => i_report_txt, i_email_cc => i_manager_list,
```

```
                i_email_extra => 'X-Priority: 1', i_env_list => 'Production,Staging');
```

If the database has access to files on the DB host, they can be converted to BLOB and attached to the email, like in this example:

```
mail.send_mail(i_email_to => 'dbas@mycompany.com', i_email_subject 'Backup Report',
                i_email_body => 'See the attached report',
                i_attach =>
io.convert_file_to_blob(l_date_prefix||'_rman_bkp.rpt','RPT_DIR'),
                i_attach_file_nm => l_date_prefix||'_rman_bkp.rpt');
```

If you have some BLOB content in memory, or stored in a column somewhere, just pass it directly via the i_attach parameter. But if your content is stored in memory or a column as a CLOB, it will need to be converted to BLOB first. And because attachments need file names, you'll need to invent a file name for it:

```
mail.send_mail(i_email_to => 'accountants@mycompany.com',
                i_email_subject 'Signature Violations!',
                i_email_body => 'The attached accounts were flagged with
signature violations.',
                i_attach => util.convert_clob_to_blob(i_sig_viol_clob),
                i_attach_file_nm => 'sig_violations.rtf');
```

The current OraJava class (used to do the email sending) cannot handle but one attachment. If your application needs to send multiple attachments for each email, MAIL will need to be re-implemented on UTL_SMTP.

Email problems detected by send_mail (usually surrounding the SMTP server) are trapped and recorded in APP_EMAIL. One can query this table to get a rich, historical account of how certain features are working.

Process Locking

Generally speaking, complex transactions are all encased within a single ACID transaction and are all committed or rolled back at the end, using underlying database technology from presenting any dirty reads. But what if that process involves things like autonomous transactions, file writes, FTP and other things that don't participate in a database transaction?

Perhaps your company has an end-of-month process that:

1. Does a calculations on the month's transactions, revisiting some tables to make adjustments.
2. Moves data to staging tables for further crunching and calculations.
3. Refreshes some materialized views.
4. Moves the processed staging data to the data warehouse.
5. Finally summarizes the results and work done and emails a copy to the board of directors.

The problem is that all the vice presidents have access and authority to run this month-end process. Rather than relying on them to play nicely together and coordinate who runs it, and when, and what to do when Joe is out of town, etc., it is better to place a control in the application or database that would prevent more than one user from kicking off the process at the same time. I call this singleton, named, or process locking.

One solution is to use ENV.tag (or DBMS_APPLICATION_INFO.set_module) to place the name of the month-end process into the database where all instances and sessions can see it. The application or backend process only needs to be coded to look for that module value before allowing itself to run.

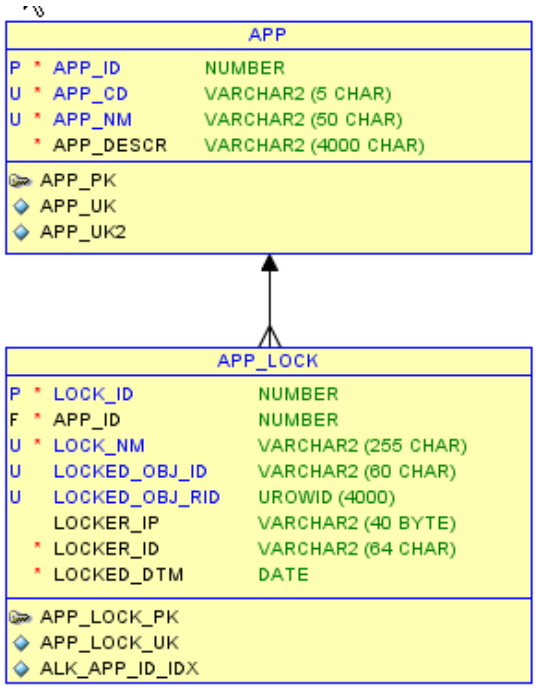
That will work, but Starter provides another solution, a table to hold each application's locks, and a simple interface in the LOCKS package to request the lock (get_lock), query the lock (read_lock) and remove the lock (del_lock). It has proved valuable on every project I've been on since Starter was built.

The advantage of the Starter solution is that the APP_LOCK table can accommodate both large-grained objects (name of process, flow, page, table, module, etc.) and fine-grained objects (PK ID or ROWID of row being locked).

The only trick to using Starter's locks is that all applications interested in a large-grained or fine-grained lock must cooperate by requesting and releasing locks, otherwise it breaks down quickly.

Typically, for backend PL/SQL processes a lock is requested and obtained by calling LOCKS.get_lock('*lock name*'). If something else already holds the lock, the LOCK_DENIED constant (CNST.false = 0) is returned, allowing the caller to respond appropriately. The caller could halt the attempt entirely, returning an error. Or they could query the lock, calling LOCKS.read_lock, which returns a PL/SQL record of every attribute in APP_LOCK for the given lock. This gives the caller good info to let the front end requester know who already has the lock, and when they obtained the lock. When the lock holder is done, they should call LOCKS.del_lock to remove their row from APP_LOCK.

See the LOCKS package spec for further details. Here is the physical data model of Starter's underlying table for named locks:



IO (File and Stdout)

To support the file-writing features of the Logging and Email libraries, the IO package was written. At the same time, a number of commonly called items in UTL_FILE were wrapped to make the IO package more complete.

If your application has a need to write to files from within the database, Starter's IO package is an excellent place to start. For most, a call to IO.write_line() or IO.write_lines() is sufficient to create a file, open it, write to it and close it. The underpinnings of the IO library handle all the complexities and possible exceptions of the UTL_FILE operations for you.

The services offered are:

Service	Short Purpose
CONVERT_FILE_TO_BLOB	BLOB. Reads a file from the file system and places the bytes into a BLOB.
COPY_FILE	Copies a file from one directory and/or name to another.
DELETE_FILE	Overlay for UTL_FILE.fremove.
FILE_EXISTS	Boolean. Uses utl_file.fgetattr to determine if a file exists.
FILE_LENGTH_BYTES	Numeric. Uses utl_file.fgetattr to determine file length in terms of bytes.
FILE_LENGTH_LINES	Numeric. Returns the number of lines in the given file.
GET_DEFAULT_DIR	Mainly used by routines in the IO package. Returns default directory used for file operations on the DB host system.
GET_DEFAULT_FILENAME	Mainly used by routines in the IO package. Returns default filename used for file operations on the DB host system.
GET_FILE_PROPS	Determines whether a given file exists and how long it is.
MOVE_FILE	Moves the given file from one location to another. You may also rename the file during the move operation.
P	Heavily overloaded replacement for DBMS_OUTPUT.put_line. Please see the package spec for full details.
READ_LINE	String. Returns the desired line from the given file.
RENAME_FILE	Overlay for UTL_FILE.frename with a couple of improvements and synergies with other IO routines and dynamic parameters.
SET_DEFAULT_DIR	Overrides the default directory used for file operations on the DB host system.
SET_DEFAULT_FILENAME	Overrides default filename used for file operations on the DB host system.
WRITE_LINE	Writes a string (<= 32K) to a file.
WRITE_LINES	Writes an array of strings to a file. Performs better than write_line().

Shared SQL

Found in the `_create_base_tables.sql` script is a commented out table named APP_SQL. Generally, system-generated dynamic SQL should be kept in the code rather than here. However, a few applications the author has worked on in the past required a table-driven set of SQL statements that could be dynamically composed. If you have a project with a similar need, uncomment this section before the install, or just manually run the table and sequence DDL after the install. Remember to perform the appropriate grants and synonyms creation in the schemas that use the framework.

The basic design of the table has each application owning its own set of rows of shared SQL. And within an application, each user can have their own set. There is a further commented-out column and foreign key which would enable groups of users within an application to share the same SQL. Really, this is just a starting point. Customize as needed.

Miscellaneous and Utility Functions

The UTIL package includes a set of constants to represent the different types of independent Oracle objects. These are primarily used by the obj_exists and attr_exists routines in the same package. If you've studied the contents of DDL_UTILS, this will look very familiar.

The UTIL package comes with a small number of miscellaneous functions that didn't seem to fit into any of the existing packages, and didn't yet warrant their own dedicated package. The routines primarily add missing IF-THEN-ELSE functions and a few conversion utilities:

Service	Short Purpose
ATTR_EXISTS	Boolean. Determines whether a contained item, like a column, type attribute or type method, is found in the containing object.
BOOL_TO_NUM	Converts a PL/SQL Boolean value to 1, 0 or NULL.
BOOL_TO_STR	Converts a PL/SQL Boolean value to "TRUE", "FALSE" or "NULL".
CONVERT_CLOB_TO_BLOB	Converts data within a CLOB to a BLOB.
GET_MAX_PK_VAL	Given a table name, retrieves the name of the PK column, and with that queries the table for the MAX(pk column) value in the table.
GET_MIME_TYPE	A simple algorithm to determine the MIME type of a file based on the file extension.
GET_OTX_DOC_TYPE	A simple algorithm to determine the Oracle Text document type for a given MIME type. Valid return values are TEXT, BINARY, and IGNORE.
IFNN	Function to perform inline if not null/then/else, giving more flexibility and enabling more elegant code. Overloaded to accommodate strings, dates and numbers.
IFNULL	Function to perform inline if null/then/else, giving more flexibility than NVL and enabling more elegant code. Overloaded to accommodate strings, dates and numbers.

ITE	Function to perform inline if/then/else, giving more flexibilty and enabling more elegant code. Overloaded to accommodate strings, dates and numbers. Now rendered moot with CASE, but was essential developing apps with the PL/SQL Web Toolkit.
NUM_TO_BOOL	Converts a numeric value to PL/SQL boolean TRUE or FALSE.
OBJ_EXISTS	Boolean. Determines if the named Oracle object exists.
RESET_SEQ	Reset's the sequence's NEXTVAL based on the current MAX value in the column populated by the sequence (useful after manual population of reference tables that will use a sequence for all future row insertions).
STR_TO_BOOL	Converts a string value to PL/SQL Boolean TRUE or FALSE.

Directory Integration

The base DDL for the SEC_USER table in `_create_base_tables.sql` has a number of columns that are commented out, columns that have a lot to do with the user's contact information. In most modern enterprises this sort of information is kept in the corporate directory server. If you have no need of a directory server, but do need to keep track of more end user contact information, either uncomment those columns and re-create the table, or create a new SEC_USER_CONTACT child table.

However, if you need to tie user entries in SEC_USER to their entry in the directory server, uncomment the ldap_dn column back in. This provides a place to hold their LDAP DN (distinguished name), which is essentially the primary key pointing to their entry in the directory server. Their DN value, and other element attributes from the directory server, can be retrieved using the included LDAP package.

The LDAP package is not installed by default since it is not as universally useful as the other features of the framework. So if your application requires integration between the database and the directory server, compile the LDAP package. Before compiling, you'll need to make a few modifications, telling the LDAP package where to find the directory server, how to login to it, whether SSL is required, etc.

Open up the ldap.pks package specification. Near the top is a constant named tree_base. Change `o=<treebase>` to the org name at the top of your directory tree.

Now open ldap.pkb. At the bottom, in the package initialization section, there are seven variables, global to the package, which must be defined. The default code assumes that you have placed the values for these variables as parameters in APP_ENV_PARM. If you do not wish to parameterize these values, you can just comment out the entire initialization block and replace it with hard-coded assignments like this:

```
g_ldap_host := 'mydirserver';
g_ldap_port := 389; -- 636 if SSL
g_ldap_bind_user := 'cn=BindAccount,ou=apps,o=mycompany';
g_ldap_bind_pswd := 'mybindpassword';
g_people_base := 'ou=People,o=mycompany';
g_wallet_path := 'file:/opt/oracle/admin/My10G/wallet/';
g_wallet_pswd := 'mywalletpassword';
```

A wallet is only required if your directory server requires SSL communication. See the Oracle documentation for how to set up wallets.

Once the parameter values are set up, and the package compiled, it is wise to first focus on getting a simple bind working. Use the ldap.test_bind routine to do this. If there are problems, turn on debugging to see where it is going wrong. Once you successfully bind to the directory server, then move on to simple and complex directory searches with filters, as seen in these examples:

Get one attribute in bulk SQL update statement, using simple ldap filter

```
UPDATE dyn_user_gtt
   SET ldap_cn = ldap.get_user_attr('employeeUserName='||user_nm, 'cn');
COMMIT;

SELECT COUNT(*)
  INTO l_count
  FROM dyn_user_gtt
 WHERE ldap_cn IS NULL;
logs.dbg('Did not find employee cn for '||l_count||' users.');
```

Get multiple attributes for a single user, using simple ldap filter

```
PROCEDURE sync_user
(
   i_user_rec IN sec_user%ROWTYPE,
   o_msg      OUT VARCHAR2
)
IS
   l_vals ldap.t_val_arr;
   ...local variables...
   l_attr_str VARCHAR2(2000) :=
'dn,cn,sn,initials,givenName,mail,telephoneNumber,ldsAccountID,preferredName,preferred
Language';

   FUNCTION get_val(i_str_idx IN VARCHAR2) RETURN VARCHAR2
   IS
   BEGIN
      RETURN l_vals(LOWER(i_str_idx));
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         RETURN NULL;
   END get_val;

BEGIN

   logs.dbg('Searching for user '||i_user_rec.user_nm);

   -- Do most common case first. Search by employeeID if found
   IF (i_user_rec.ldap_uid IS NOT NULL) THEN
      -- existing user
```

```
      l_vals :=
ldap.get_user_attrs2('(employeeID='||i_user_rec.ldap_uid||')',l_attr_str);

      IF (l_vals.count = 0) THEN
         -- existing user no longer found in LDS Account
         o_msg := 'User '||i_user_rec.user_nm||' has invalid employeeID
['||i_user_rec.ldap_uid||']. No longer found in Corporate Dir Server.'||CHR(10);
      END IF;
   ELSE
      -- new user, or user that doesn't have employeeID in Corporate Dir Server
      l_vals := ldap.get_user_attrs2('(cn='||i_user_rec.user_nm||')',l_attr_str);

      IF (l_vals.count = 0) THEN
         -- user not found in LDS Account by that CN
         o_msg := 'User '||i_user_rec.user_nm||' not found in Corporate Dir Server.
Either no longer employed or login ID has spelling errors.'||CHR(10);
      END IF;
   END IF;


   IF (l_vals.count > 0) THEN
      -- assign results of LDAP query to local variables to update SEC_USER
      -- if there have been changes
      l_ldap_dn := get_val('dn');
      l_user_nm := get_val('cn');
      l_last_nm := get_val('sn');
      l_middle_nm := get_val('initials');
      l_first_nm := get_val('givenName');
      l_pmy_email_address := get_val('mail');
      l_pmy_phone := get_val('telephoneNumber');
      l_ldap_uid := get_val('employeeID');

   ... snipped much code that compares and updates SEC_USER...
END sync_user;
```

There are other routines exposed publicly in the package spec, for added flexibility, but the primary routines meant to be used are get_user_attr, get_user_attrs and get_user_attrs2. The first gets a single attribute. The second returns a collection (array) of attribute name/value pairs. The third returns a string-indexed array of values, making it much easier replicate random access into the user's directory server attributes.

Unit Testing

Early versions of Starter (pre-open source) came with a home-grown package that attempted to encourage test-first development and repeatable test-suites. Steven Feuerstein started developing utPLSQL shortly after that, which quickly took on a life of its own. So the Starter test framework languished and was eventually dropped.

Nevertheless, a good testing framework is still essential to developing on any platform. Your shop absolutely should evaluate and set one up. Take a look at utPLSQL, but especially Feuerstein's later effort: Quest Code Tester. It does cost a bundle, but may be worth it for your needs. Another option to evaluate is PLUTO, an open-source PL/SQL testing framework from a developer at Google.

Database Code Templates

An application development framework isn't complete without a set of templates for the common objects created. The sample PL/SQL programming standard document included in the Standards folder of the Starter framework includes sample templates that you may use or modify for your needs.

If you use PL/SQL Developer as your IDE, email Starter's author for a copy of his .tpl files which codify the templates mentioned above.

Database Naming Standard

Although naming seems trivial when quibbling over a prefixes, abbreviations, a few extra characters here and there, it becomes extremely important years down the road. Applications tend to grow. Business changes. Table names and functional areas tend to expand and lengthen. Next thing you know, you're bumping up against the infamous limitation of 30 characters for Oracle object names. You start arbitrarily truncating and abbreviating name tokens to get the new, lengthier object names to fit, and you wish you'd abbreviated and had a clear naming scheme years before when it wouldn't have cost the company nearly as much to refactor and make the changes.

So a good naming scheme from the start of a project, or even a company's inception, is a best practice and great idea. Starter includes a sample Naming standard that is copyrighted by the author. You may use it and modify for your internal needs, but please don't attempt to sell your version. It is part of open-source software after all; such a thing would violate the nature of OSS.

The name of the document is `\Standards\Oracle Naming Standard.pdf.`

There is also a companion spreadsheet, `\Standards\Abbreviations and Acronyms.xlsx`, which gives you a head start on an abbreviation standard for your company.

Database Programming Standard

Any coder worth their salt knows that standards are a great thing, both short and long term. The Agile family of development practices (especially XP) speak of this explicitly. When developers follow some rules and guidelines, they produce code that looks the same, compiles the first time, generally passes most tests the first time and is a joy to own and maintain. Following standards reduces fear and enables anyone on the team to work on the code produced by others. There are myriad other benefits to clean, standardized code, but this really isn't the time or place to convince the unconvinced.

For those looking for a jumpstart to their PL/SQL development, Starter also includes a copyrighted sample PL/SQL programming standards doc that, again, you may use and modify, but please don't attempt to sell your version.

The name of the document is `\Standards\PLSQL Programming Standard.docx`.

There is a companion document that can assist Oracle database architects and development DBAs in their design of performant database schemas. It is named `\Standards\Oracle DB Design Guide.pdf`.