Dynamic Debugging

and

Instrumentation of Production PL/SQL

Bill Coulam, DBArtisans.com

A day in the life of a DBA

1. The CFO sends an irate email to the CIO, who calls your manager. It seems the finance department has been attempting to run the monthly reconciliation and transfer, but it is not returning. It’s been running for 3 hours, but normally only takes 45 minutes. It’s 2pm on Friday, and Chicago needs the figures before 3pm. All eyes are on you.
2. Your employer owns a chain of car dealerships, and it has come to his attention that at least one of his sales managers has been altering the sales contracts after entry into the system, adjusting the figures for the new car and trade-in, thus boosting the commission but keeping the overall contract the same amount. The owner wants to know who else has used their privileged account to pad their pockets in this manner.
3. Emails and calls are overwhelming your firm’s CRM system. The busy dealer and customer websites are extremely sluggish, work is backing up, and tempers are flaring as the company loses about $3K per minute. Unfortunately, the database was the culprit about a year ago, so all fingers are pointing at you again. Guilty until proven innocent…
4. Most of the 4,000 sales reps for your telecom company are working just fine, but one rep in particular, Ana B., hasn’t been able to use the APEX-based Sales and Promotions application in a day. Ana’s supervisor got past support and found your number. She wants to know help Ana get back to work ASAP.

How do you handle scenarios like this? Will it be a few minutes until you have the answer, or are you cancelling those weekend getaway plans?

Rather than paint a grim picture of how the above scenarios will play out in the typical Oracle shop, this paper will describe how smooth things could be if code is instrumented and show how it is done. We’ll also briefly survey existing Oracle built-ins and open-source instrumentation libraries, building a list of minimal requirements you can employ to evaluate them or design your own. It is hoped the reader will catch the vision, download a library, install it and begin using it this very day.

Instrumentation, for fun and profit

Instrumentation is the process of fitting production applications with code that directs runtime context to some destination where it can be useful.

The runtime context can be who, when, arguments, changes, elapsed time, remaining workload, errors, warnings, etc. These useful bits of runtime insight can be lumped into three categories of instrumentation:

**Debugging** – Lots of highly detailed messages that, when enabled, can show a troubleshooter the exact path a call took and all the parameter and variable values experienced during execution. Due to sheer volume, debug messages are typically disabled in production code, but can be dynamically enabled by changing a table-based key:value pair.

**Logging** – Permanently enabled messages generated by the code at runtime, logging valuable information the business may require. This information is typically larger-grained than the detailed debug messages, such as when a critical process started and stopped and how long it took (metrics), or warnings and errors detected and the values being processed at the time of the error.

**Auditing** – Data that exposes who changed what and when it was done. Although Oracle comes with many built-in auditing features, it still does not offer column-level auditing except through 11g’s extra cost FDA option. For the rest of us, the solution is per-table triggers that record who, when and what, the “what” being old and new column values upon DML events. The audit data can be mixed in with the debug and log messages, but I’ve found it more useful to keep them separate in a table set designed for tracking changes to rows and columns.

The destination of the runtime context can be stdout, V$ views, a logging table, a log file on the database host, a queue for asynchronous publication and monitoring, a DBMS pipe or alert, and other slower, more complex alternatives like HTTP, FTP and UDP callouts. Most architects prefer logging within an anonymous transaction to a logging table, which immediately imparts all of the database’s benefits upon it, e.g. it becomes sortable, mineable, partition-able, archive-able, recoverable, shareable and so on, all of which are harder or impossible when sending messages to files or other destinations.

Instrumentation 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.

Instrumentation is a best practice in any programming language, and one of the clearest indicators of a mature software engineer, or artisan. It’s easy, requires very little overhead, and will prove you to be a visionary and hero over and over as life, such as it is, happens to you and your company.

The same day, instrumented

Let’s look at how addressing the situations in the first section can be passably pleasant when instrumentation is in place.

1. In the first scenario, you open V$SESSION and run your favorite query. Because the instrumentation took advantage of DBMS\_APPLICATION\_INFO, you can immediately see the active row that pertains to the finance process, because module, action and client\_info are filled and show you the very package and routine where the process is at. Because you follow modular coding practices, that routine only does one thing, which is the update it is stuck on. How long does it normally take? A quick query against the logging table for that routine’s metrics shows it consistently takes only a few seconds. You re-query V$SESSION and V$SESS\_IO, but very little changes. It is probably waiting on something. Wait statistics can be great, but you run another query to check for DML locks. Sure enough, there is an uncommitted transaction against that table, coming from the CFO’s desktop. A quick call confirms the now-contrite CFO forgot to commit or rollback a large update of a new column on the ledger before putting his machine to sleep to catch a cab to the airport. With permission, you kill his session, and the process completes a few minutes later.
2. In the second scenario you run a stored query against the column-level change audit table, looking for all changes to the contract tables. Within minutes you see the balancing updates coming from two sales managers, brothers actually, logged from their office workstations right after closing time each Saturday night.
3. In the third scenario, you calmly roll up your sleeves, and then attack the keys with the passion of a concert pianist, crafting a query against the logging table, looking for average response times for all modules over the last month. You run the same query but just for the last day and hour. In general, you see the response times of the calls into the database routines are unchanged. Phew! These results are emailed to the general manager who now sends the attack dogs after the network and app server administrators.
4. In the last scenario, the pressure isn’t as great, but how does one debug the path of a single sales rep within all the stored PL/SQL that 3,999 other reps share and hit pretty hard, every minute of every day, concurrently. Easy! You get the rep’s login ID and change the Debug value in the parameter table from “off” to “user=ana.b”. Since the frontend application cooperates with the backend, the frontend user is identified upon every call into the database. As soon as Ana tries again what has been failing for her, the existing debug lines, peppered all over your production code, “wake up” and start logging detailed runtime context only for Ana. Within 10 seconds of observing incoming rows in the logging table, you see the problem; it was a permissions issue in the company directory server, as her entry was bungled by the previous night’s directory replication. The DBMS\_LDAP call was binding, but returning no attributes, which wasn’t an error condition, but not something the PL/SQL authorization service was defensively coded to handle. A quick call to the networking team fixes that and she’s off and running again. Ana is happy. Her manager is happy. Your manager is glowing from the praise being heaped upon her. You set the Debug toggle to “off” again and go back to a well-deserved lunchtime break, reading up on programming practices of the pros.

Good instrumentation, like good security and thorough testing, is never yearned for until it is far too late. The scenarios above are quite typical for any of us with a significant investment in Oracle and PL/SQL. Inevitably something will go wrong. When it does, instrumentation often means you’ll be spending a few minutes on the problem instead of hours, days, or simply having to admit you can’t get the information needed.

We’ll now look at a few built-in instrumentation features in Oracle and their limitations. This will inform the list of requirements for a good instrumentation framework

Existing instrumentation features in Oracle

Oracle offers a few built-in packages and tools for debugging and real-time insight into active sessions, but not logging or column-level auditing. There is not enough room to be exhaustive, so we’ll briefly consider their advantages and limitations. Attendees of the presentation will enjoy a short demo and discussion of each.

Column-level auditing

Oracle’s basic and fine-grained auditing features *seem* promising, but it turns out that neither provide column-level auditing that can track who changed a column value, what changed, and when. With every new version of Oracle I check the new features to see if column-level auditing is finally addressed. This anticipation was finally answered with Flashback Data Archive (aka Total Recall), introduced in 11g. This is an efficient method of tracking the changes to data over time and does not require massive undo space as did the older Flashback Query. However, this is a rather expensive extra cost option, and again takes considerable effort to get the who, what and when into a friendly format.

So for most of us, Oracle effectively still does not have a built-in feature for column level auditing. We must build our own, which is, thankfully, not hard to do.

Metrics

DBMS\_UTILITY

DBMS\_UTILITY.get\_time returns elapsed time in terms of hundredths of seconds (from an arbitrary point in time known internally to Oracle). Calling it once, doing processing, then calling it again, and subtracting the first from the second value gives us elapsed time down to hundredth of second granularity. If seconds is sufficient, divide the result by 100.

Without DBMS\_UTILITY.get\_time, fine-grained metrics would be difficult or impossible for PL/SQL programs. Although one can write custom code to utilize DBMS\_UTILITY.get\_time every time metrics are desired (which should be everywhere), it is better to wrap its use in a logging/metrics framework API, so that timings are derived consistently

Debugging and Logging

When most Oracle developers think of debugging PL/SQL, their thoughts turn to DBMS\_OUTPUT. This is like bringing the rusty, flatted BMX bike you had as a kid to a street race, instead of the Yamaha YZF-R1 superbike sitting unused in your garage. DBMS\_OUTPUT, like Java’s System.out.println, is known as a “poor man’s” debugger. Although it can be useful, and has been a staple of PL/SQL coders for 15 years, it should only be used in quick-and-dirty development and discovery. It has many limitations and far superior alternatives, like DBMS\_PIPE and DBMS\_DEBUG, which we’ll cover…now.

DBMS\_DEBUG and debugging IDEs

Traditional debugging is usually done within a programmer’s IDE and allows the troubleshooter to step in/out of modules, run to breakpoint or condition or exception, add watches, change variables at runtime, peer into memory structures, view code execution in real-time, etc. PL/SQL has this capability too.

Since Oracle 7, there has been a debugging API that PL/SQL IDEs have used to provide the veritable debugging superbike. In Oracle 11g, this API is embodied in DBMS\_DEBUG and DBMS\_DEBUG\_JDWP. The developer’s account must have DEBUG CONNECT SESSION system privilege, but it appears that DEBUG ANY PROCEDURE is no longer needed unless debugging objects in others’ accounts. The developer uses her IDE to compile the package or trigger with debug info[[1]](#footnote-1), places a breakpoint in the code object, and uses the IDE to initiate a debugging run of the stored object. With DBMS\_DEBUG\_JDWP it is possible to begin a session with some other application, and have the IDE “wake up” when it detects a connection entering that object. It is even possible to debug like this in Production if code is permanently or temporarily compiled for debug. However, leaving objects compiled for debug in Production is not recommended. Performance can be impacted due to the overhead it imposes. But in my experience, the scariest thing was stability. On rare occasions I’ve had sessions hang, and hang in really nasty ways when debugging. The killing and cleanup of the affected sessions was tricky. This is not something you want to have happen in Production.

Personally I only use IDE debugging in development to hunt down the rare infinite loop, and to force certain codepaths for full coverage unit testing. In general it works very well there have been issues surrounding the debug API and its interactions with connection pools, listeners, RAC databases, firewalls and database version/host OS bugs. Most of these are now fixed, but is has left me a bit jaded. The biggest weakness of this feature is that it does no good if the client application was not coded for remote debugging. It is impossible to connect a debugger to someone’s active, problematic session, or even worse, sub-second session connections. In these cases the solution is to have the sessions report their own codepath and state with debugging messages. Since debugging could occupy an entire article on its own, I’ll cut it short here. Just know that IDE debugging of PL/SQL programs should be a tool in your bag of tricks.

ORADEBUG

This “undocumented” but well-known utility does allow real-time peeking into other sessions, one of the things instrumentation should be able to do. Unfortunately, it is oriented more towards really low-level memory and process debugging and tracing. It is the stuff of Oracle wizards that peep and mutter. Oracle Support prefers that you not use it unless instructed to. That it requires SYSDBA privilege is another factor against using it for production instrumentation. There are many informative papers on this utility if still curious, but there are definitely far less obtuse ways of peering into the execution of active sessions.

DBMS\_PROFILER

Like Oracle’s debug feature, the Profiler is wonderful, especially in an IDE like SQL Developer or TOAD that wraps the profiler in nice graphs, immediately pointing out hot spots and time hogs. However, like the debug feature, it is less useful for the sessions and routines being accessed by end users. It is better used while tuning in development where you have full control of the start of the session and what is called.

DBMS\_ERRLOG

This package sure sounds like what we need for logging, but the name is somewhat misleading. It is only used to create a special DML error logging table. It is useful in its own right, but useless for the kind of logging we need.

DBMS\_ALERT

Despite the docs indicating DBMS\_ALERT is useful for asynchronous notification of database events, its use is transactional; that is the waiting client can’t see the desired event message until after the alerting session commits. This is an Achilles heel for instrumentation which needs to deliver its messages, even if the transaction fails and rolls back.

According to various sources, using alerts is also resource intensive. The client has to wait (blocks) and the server piece requires a pipe and a lock. Writing the event waiter in PL/SQL is problematic, unless the event message is written to a file or table via anonymous transaction to gain immediate visibility. Another option is to write an alert event listener in some other language that is not hindered by the limitations of transactions. PL/SQL Developer has an event monitor just like this. Unfortunately, I found that it takes about a second to get registered and return to waiting, as the signals that followed right after a prior signal were simply lost. Oracle docs warn about this possibility. The messages are limited to 1800 chars as well. Finally, session-specific metadata, like client\_id isn’t communicated across to another session. Taken together, DBMS\_ALERT is unsuitable for instrumentation.

DBMS\_PIPE

DBMS\_PIPE is actually promising. Sending messages is independent of the sending session’s transaction. There is a level of security offered with private pipes that could be perfect for debug, timing and error logging within the application object-owning account. Packing the messages is a little cumbersome, and it does not guarantee message delivery like AQ does. Furthermore, once the message is consumed it is automatically removed from the buffer and cannot be read again. One can send the message to a table or file where it can be read again, but that begs the question why the message wasn’t sent directly to the table in the first place, bypassing the pipe entirely? DBMS\_PIPE is a viable piece of infrastructure for getting instrumentation messages out of an application.

DBMS\_APPLICATION\_INFO

There are a few pieces of client metadata that, when set, are available in V$SESSION and a few other performance views. Since many performance views can join to V$SESSION by means of the sid+serial# or sqlid, availability in V$SESSION is typically sufficient. This metadata is labeled as module, action, client\_info and client\_identifier. Despite what the package specification says in 11g, Oracle still truncates their length to 48, 32, 64 and 64 bytes respectively. They are somewhat weak individually, but in combination they are powerful. Originally intended for 2-tier and client-server applications to identify themselves to the database, they can be put to great use inside PL/SQL programs to provide DBAs with low-overhead, highly useful, real-time, transaction-independent keyhole views into what the program is currently doing. This is particularly handy when debugging programs taking longer than expected or hanging. I like to call this “tagging” a session.

DBMS\_APPLICATION\_INFO’s main routines are set\_module(), set\_client\_info() and set\_session\_longops(). Set module *and* action with set\_module(). Track lower level process detail with set\_client\_info(). To set the client\_identifier, aka client\_id, use DBMS\_SESSION.set\_identifier. Client\_id is critical. It is used by authentication and security schemes, basic auditing, column-level auditing, tracing with DBMS\_MONITOR and more. Ensure your frontend applications are passing the user’s login ID to the database to be stored in the client\_identifier[[2]](#footnote-2).

On certain resource intensive operations (like DML on more than 10,000 blocks) and certain parallel operations, Oracle automatically records how much work it has to do and how far along it is in the V$SESSION\_LONGOPS view. With this info one can construct a query and even a frontend progress bar to inform how long database operations will take[[3]](#footnote-3). Oracle allows us to write to that view to track our own scripts, DDL operations or DML statements through set\_session\_longops().

Drawbacks to calling DBMS\_APPLICATION\_INFO routines pertain to the durability of the tags. Sometimes they stay around too long (if the developer forgets to clear them out[[4]](#footnote-4)). This is especially risky with the client\_identifier, which could accuse the wrong user as the changer of sensitive data. Sometimes they are cleared or overwritten prematurely. This is particularly tricky if one instrumented routine with tags calls another instrumented routine with tags. This wipes out the session tag from the calling routine, leaving the incorrect tags in place once control returns from the subroutine. So although DBMS\_APPLICATION\_INFO should be an integral part of an instrumentation library, it is no fun to type repeatedly and it should be wrapped in a library to nested tagging.

DBMS\_OUTPUT

As mentioned earlier, DBMS\_OUTPUT is often used to slowly, excruciatingly debug PL/SQL. Since characters placed in the buffer must be retrieved by a tool like SQL\*Plus and TOAD to be useful, using DBMS\_OUTPUT in Production is a wasteful and useless. Essentially anything important sent to DBMS\_OUTPUT in Production is lost. However, the biggest limitation is that the buffer cannot be retrieved until after the code has executed, which renders it useless for real-time insight into program execution and state.

A more appropriate use of DBMS\_OUTPUT is as a quick-and-dirty logging tool for transient anonymous blocks, often written during data exploration, unit testing and development. We use DBMS\_OUTPUT within our automated database build system to read what happened in scripts that had anonymous PL/SQL blocks, piping the output to the build logs, which are further examined by the tool for build success or failure. If building, buying or adopting an instrumentation library, ensure that it includes the ability to output messages to stdout/screen, and that it is transaction-independent.

UTL\_FILE

The UTL\_FILE package provides the low-level framework required of any logging solution that wishes to write to database host files. As long as a file has been successfully opened in write or append mode, calling UTL\_FILE.put\_line will send a message to the file, formatted as desired (up to 32K characters per line), independent of the encompassing transaction. However, using all of UTL\_FILE’s constants, exceptions and routines is rather involved and prone to human error. It is best to wrap this in a custom file-logging API that hides much of the complexity for your developers.

DBMS\_SYSTEM

SYS.DBMS\_SYSTEM includes the ksdwrt() routine, which lets you write messages directly to the alert log, independent of the containing transaction[[5]](#footnote-5). It is the closest thing Oracle includes that almost matches our needs for logging. If the client identifier has been set, it will be used in the alert log entry, helping pinpoint *who* generated a particular message. A timestamp will be written, along with other session metadata like client module (program), client machine name and address, module, and host process ID. This is what a custom ksdwrt alert log message looks like:

<msg time='2012-02-03T18:30:40.283-07:00' org\_id='oracle' comp\_id='rdbms'

client\_id='bcoulam' type='UNKNOWN' level='16'

host\_id='R9AXR65' host\_addr='fe80::cd94:25d3:ee1a:9777%11' module='PL/SQL Developer'

pid='15156'>

<txt>WARNING! Here is my real-time msg logged to alert.log

</txt>

</msg>

DBMS\_SYSTEM is not typically granted to non-SYSDBA accounts (for good reason). Writing to the alert log is not a great idea either. Oddly, each new line in your message is interpreted by ksdwrt as a separate message, and normal characters are escaped with their HTML equivalents, making some messages to read outside of a browser. Plus you can’t control the format of the log messages. Think very carefully before opening its use up to other accounts or roles.

Metadata Gathering Routines

There are a number of scattered helper routines and built-in functions that can return metatadata about the connected client, database host, database, instance, version, etc. These are things like DBMS\_UTILITY.current\_instance, DBMS\_DB\_VERSION.version and release, and [SYS\_CONTEXT(‘USERENV’,’<attribute>’)](http://docs.oracle.com/cd/E11882_01/server.112/e26088/functions184.htm#SQLRF06117) which offers a host of values describing the current session. These should be included in your instrumentation library so that they get used and used consistently when logging messages.

Requirements of an instrumentation library

After reviewing some typical emergency scenarios and the limitations of built-in Oracle instrumentation, a clearer picture of instrumentation requirements begins to form.

1. **Simple API to measure elapsed time.** Ideally allow multiple running timers and nested timers (in case one instrumented routine calls another).
2. **Simple API to log a message that is independent from the calling transaction.** Must be configurable to output to the screen and a logging table. Good if file output is also offered. Output to named DBMS\_PIPE or FTP server is nice-to-have, but not essential. Allow different types of messages so that debug messages are ignored until requested. Excellent if debug messages can be enabled for a given user, PL/SQL object, named business process or active session.
3. **Standardized, simple method to log and handle exceptions** and user-defined error messages.
4. **Simple API to write to files on the host system.** Should be able to write many lines at a time to aid performance.
5. **Simple API to tag/untag sessions and track long operations.** Ideally allow nested tagging (in case one instrumented routine calls another).
6. **Some aids to ease the creation of column-level audit tables and triggers**.
7. **Routines to simplify the retrieval of session, client and database metadata.** Meant to be used independently and transparently by the logging API, but can be called directly as well.

With these requirements satisfied, and the simple APIs used consistently, it should be child’s play to examine the audit and logging tables, and enable debugging, when things go wrong, to immediately see:

* who called the routine, their IP address, client machine name, and client OS user name
* when they called the routine, how long the subroutines took, and how long the entire call took
* what the call accomplished, the number of iterations, the current iteration, the value(s) passed back
* parameter values passed into the routine, as well as changing and derived variables determined within
* old and new column values if data is changing
* path the call took through the control statements
* anomalies, warnings and errors detected, etc.

Wouldn’t that be nice?

Existing Instrumentation libraries

Before designing a proprietary instrumentation library, it is a good idea to survey the current market and determine if any already satisfy requirements. Ten years ago, there were about two choices. Today there are a number to choose from. Those that claim to be full PL/SQL application frameworks are lightly shaded.

|  |  |  |  |
| --- | --- | --- | --- |
| **Resource Name** | **License** | **Purpose** | **Location & Notes** |
| Google Code | Free | Library of libraries | <http://code.google.com/hosting/search?q=label:plsql> |
| Feuerstein  PL/SQL Obsession | Free | Repository of all things SF and PL/SQL | <http://www.toadworld.com/sf> |
| QCGU (Quest CodeGen Utility) | Free | Full framework, Standards, Scripts, Template Factory, Code Generation, + more | <http://codegen.inside.quest.com/index.jspa>  Latest incarnation of Feuerstein's vast reservoir of experience. (successor of QXNO, PL/Vision, and PL/Generator.) |
| PL/SQL Starter | Free | Author's full framework. | <http://sourceforge.net/projects/plsqlframestart> |
| Simple Starter | Free | Logging, Timing, Auditing, Debugging, Error Handling, + more | Simplified PL/SQL Starter to just logging, timing and auditing components (and the low-level packages they depend on). Designed to be used in one schema. Install and begin using in under a minute. |
| GED Toolkit | $120-$1200 | Almost full framework | <http://gedtoolkit.com>  Includes APEX UI to administer jobs and tables. Monitor processing. |
| PL/Vision | Free | Framework, API Generator, + more | <http://toadworld.com/Downloads/PLVisionFreeware/tabid/687/Default.aspx>  Replaced by QXNO and then QCGU. Not supported. |
| Log4ora | Free | Logging | <http://code.google.com/p/log4ora/>  Fresh, full-featured logging library. Alerts. AQ. Easy to use. Good stuff. |
| ILO | Free | Timing and Tuning | <http://sourceforge.net/projects/ilo>  From the sharp minds at Hotsos |
| Quest Error Manager | Free | Error Handling | <http://www.toadworld.com/LinkClick.aspx?link=685&tabid=153>  Included in QCGU. But offered separately as well. Not supported. |
| Plsql-commons | Free | Collection of utilities, including logging | <http://code.google.com/p/plsql-commons> |
| Log4oracle-plsql | Free | Logging | [http://code.google.com/p/log4oracle-plsql](http://code.google.com/p/log4oracle-plsql/)  Seems like an active project, but could not find code to download… |
| Log4PLSQL | Free | Logging | <http://sourceforge.net/projects/log4plsql>  Popular, but aging and complex log4j analog in PL/SQL |
| Logger | Free | Logging | <http://sn.im/logger1.4>  Recently orphaned when Oracle decommissioned its samplecode site. Simple. Easy to use. |
| Orate | Free | Logging | <http://sourceforge.net/projects/orate>  Never used it, but has been around a while. Still active. |

Many of the above are quite good. Some are limited to just logging. Some are more complex to use than others. In my decidedly biased opinion, the easiest one to begin using, which satisfies the requirements on the previous page, is the “Simple” version of the open-source PL/SQL Starter Framework. The remainder of this paper will walk through the Simple Starter model, how to install and configure it, and provide examples of adding instrumentation to production code. Although this paper will demonstrate only one framework, it is hoped the reader will evaluate the other libraries for their merits and catch the vision of how easy it is to use any of them to add maturity and maintainability to applications.

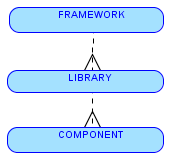
PL/SQL “Starter” Framework: the simple[r] version

This seasoned PL/SQL framework has been covered in previous presentations at RMOUG. It is a standard application infrastructure which can support many applications resident in different schemas on the same database. Even though it has been simplified as much as possible, due to its flexibility and set of 21 major services, it still requires 55 objects to operate and a 60 page document to explain in full. For this presentation, and for those who only want to use instrumentation in a single schema, the framework was slimmed down again to 17 objects explicitly offering seven major services. The seven libraries include those needed for instrumentation: Auditing, Metrics, Debugging/Logging and some “extras” like Standardized Error Handling, File Operations and Connection Metadata (without which the first four wouldn’t function).

Installation and Configuration

Go to SourceForge and search for PL/SQL Framework. It should be the first result, next to an icon depicting a warm loaf of bread raised with sourdough starter (yes, it was yummy). Click through to the project’s homepage. Find and click “Browse all files”. Download the zip file with “Simple” in the label. Unzip it. Start SQL\*Plus or a Command window in PL/SQL Developer as SYS and run *\_\_InstallSimpleFmwk.sql*. You can choose an existing schema to install it to, or let it create the default, the APP schema. It will run you through a few questions, and then create the objects it needs to function. That’s it! There is no further configuration required. If any of the choices weren’t quite right, drop the schema and re-install. Some of the choices are recorded in the APP\_PARM table. You may adjust these as you see fit if directories or subfolders change.

Installed objects

Frameworks are collections of related libraries, and *libraries* are collections of related components. A *component* is the finest-grain building block upon which a framework-based application is built. Logically this relationship looks like the model to the right:

Applied to database application frameworks, the component is implemented as an Oracle object, including triggers, types, tables, views and routines (I use the generic term *routine* when referring to a packaged function or procedure.) The library will often present its interface within a PL/SQL package. And the framework is the entire collection of packages and all the components of each.

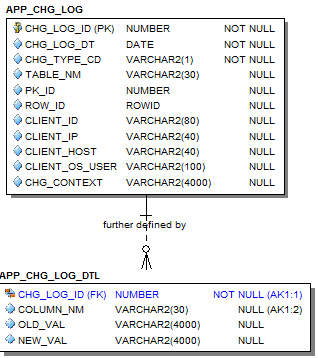
Within the Simple Starter Framework, one finds the following libraries. Although it is good to get familiar with all the components and read the extensive comments in the package specifications, one only need study the items below in bold font to prepare to instrument:

|  |  |  |
| --- | --- | --- |
| **Library** | **Main Routines** | **Supporting Objects and Notes** |
| Auditing:  **gen\_audit\_triggers.sql** |  | APP\_CHG\_LOG, APP\_CHG\_LOG\_DTL (tables) |
| Metrics:  **TIMER** (package) | **startme()**  **stopme()**  **elapsed()** | DBMS\_UTILITY |
| Debugging, Logging and Error Handling:  **LOGS** (package)  EXCP (package meant to be used only by LOGS)  APP\_LOG\_API (pkg meant to be used only by LOGS) | **err()**  **warn()**  **info()**  **dbg()** | APP\_LOG (table)  TRIM\_APP\_LOG (scheduled job) |
| Connection Metadata:  **ENV** (package) | **init/reset\_client\_ctx()**  **tag/untag()**  **tag\_longop()** | DBMS\_DB\_VERSION, DBMS\_APPLICATION\_INFO, DBMS\_SYSTEM, v$session and v$mystat. |
| File Operations:  IO (package meant to be used primarily by LOGS) | write\_line()  write\_lines(0  p() | UTL\_FILE, DBMS\_LOB |
| Dynamic (Table-Driven) Parameters/Properties:  PARM (package) | get\_val() | APP\_PARM (table) |
| Extras (required for the seven libraries above to function):  CNST, TYP, DDL\_UTILS,  DT, STR, NUM (packages) | These are libraries of application-wide constants and subtypes, build utility functions; date, string and number manipulation routines. | |

Column-level auditing

There are at least six different ways to record changes to data over time. Simple 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 columns are identified that must be audited -- in particular financial and legal data -- it is very easy to start auditing them. Starter includes a script named *gen\_audit\_triggers.sql*. As the table-owning schema, run this script. It will output an audit trigger to a spooled file for each table[[6]](#footnote-6) 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 APP\_CHG\_LOG tables.

The best part about this 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, which IP and box 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 to retrieve and/or flatten the data into a single, more easily-read 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 new 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. The additional DDL to partition is commented out in the install script. 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 range partitioning key.

Metrics

It is said that one cannot manage that which is not measured. If all of the 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 timing routines are kept in the TIMER package, which includes three simple routines: startme(), stopme() and the function 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 lets us 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 using LOGS.info(), or dynamically by embedding them in LOGS.dbg() 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 of TIMER in action:

timer.startme('outer loop');

FOR l\_rec IN cur\_load\_waiting LOOP

timer.startme('load');

loader.process(l\_rec);

timer.stopme('load');

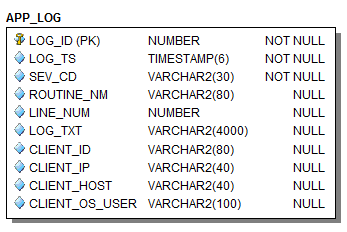
logs.dbg('Processing load ID '||l\_rec.id||' took '||timer.elapsed('load')||' seconds.');

END LOOP;

timer.stopme('outer loop');

logs.dbg('Loading took '||timer.elapsed('outer loop')||' seconds.');

Debugging and Logging

The library for logging, debugging and recording metrics is the LOGS[[7]](#footnote-7) package. Simple Starter allows one to log to the file system, a table, or the screen. It defaults to table and screen. The output destinations can be altered through the parameter *Default Log Targets* in APP\_PARM, or dynamically through logs.set\_log\_targets(). I find logging to the screen useful in development, and logging to a generic logging table useful for all other environments.

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 ENV is being used correctly.

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.

If your code has some known exceptions it needs to trap and log, capture the known errors by exception name[[8]](#footnote-8). Do any necessary cleanup (like explicitly closing files and 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.

LOGS.err() can be called with no arguments. It will derive the exception from SQLERRM, log it, and re-raise the error. But most will want to include a little runtime context (parameters and variables) in the message being logged along with the error. Pass this into the first parameter of LOGS.err(). All the other metadata surrounding the client session, the package name, routine name, line number, user ID, and timestamp will be automatically recorded for you.

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](http://www.google.com/search?q=Steven+Feuerstein+exception+OR+error+handling) on this subject for the reasons behind this statement. Using LOGS.err() fulfills that best practice.

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 a historical data trail indicating how long some process took to run, which files were available at the time of process initialization, which organization is no longer found in the master data, etc. Anything you want permanently and immediately recorded about the session can use LOGS.info() to get a message stored into APP\_LOG. This is the routine used to record timing metrics as well.

Debugging

The debugging component of the logging library is table-driven. It is meant to be dynamic, where you can turn on debug logging in Production (without compiling or invalidating anything) for a specific session, package, named process, job, schema or client identifier. With the Debug toggle turned off, all calls to logs.dbg consume miniscule overhead and are not observed. But when the toggle is on, the strings and variables passed to logs.dbg are logged to the current log targets for the application (see *Default Log Targets* in APP\_PARM).

To turn it on for all objects, you simply update the value of the *Debug* parameter in APP\_PARM to any case-insensitive positive string (on, y, Y, yes, TRUE, etc.). To turn it on for a user, update the parameter to “user=*userID*” where the userID matches the client ID passed to ENV.init\_client\_ctx by the application. Or the parameter can be “unit=*pl/sql object name(s)*” to observe debug messages only for the named object, or comma-separated list of objects. Or if you need to enable debug messages for an active session that may be having issues, update the parameter to “session=*sessionID.”*

logs.dbg('Processing salesperson '||l\_rec.emp\_name||'. Iteration '||l\_idx);

When I code, I first write the public interface's comment block, then the interface, then the tests, then the implementation. When coding the implementation, I first use pseudo-code in PL/SQL comments to outline the algorithm. Later, as I'm filling in the code I wrap the pseudo-code, along with context (like parameters and local variables) into calls to logs.dbg. This *accomplishes three things at the same time*: comments the code, provides built-in debug calls that can remain in production code, and enriches the debug calls with dynamic runtime context (something regular PL/SQL comments cannot do).

Monitoring, Tracing and Troubleshooting

A subset of the routines in the ENV package set the previously mentioned module, action and client\_info values into session performance metadata and active session history. They are a wonderful way of providing real-time, dynamic views into what your program is actually working on. 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. ENV 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();

ENV also now includes a routine that makes it smooth and simple to track long operations in the v$session\_longops view, called ENV.tag\_longop(). Examine this snippet:

**PROCEDURE** upd\_all\_initcap **IS**

l\_longop env.t\_longop;

**BEGIN**

...

-- count needed for longops tracking

**SELECT COUNT**(\*) **INTO** l\_longop.total\_work **FROM** big\_table;

l\_longop.op\_nm := 'loop and update initcap';

l\_longop.units\_of\_measure := 'rows updated.';

l\_longop.work\_target := 'BIG\_TABLE';

env.tag\_longop(l\_longop); -- seed row, then do the operation

**FOR** lr **IN** (**SELECT** \* **FROM** big\_table) **LOOP**

l\_longop.work\_done := l\_longop.work\_done + 1;

**UPDATE** big\_table

**SET** NAME = **INITCAP**(NAME)

,mod\_by = **SYS\_CONTEXT**('userenv', 'client\_identifier')

,mod\_dt = **SYSDATE**

**WHERE** ID = lr.id;

env.tag\_longop(l\_longop); -- keep v$session\_longops informed

**END** **LOOP**;

...

**END** upd\_all\_initcap;

Putting it all together

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 reports

AS

rpt\_div\_line CONSTANT VARCHAR2(80) := RPAD('\*',80,'\*');

*-- Pass an email address if on non-Prod*

PROCEDURE print\_and\_send\_ps(i\_email\_addr IN VARCHAR2 DEFAULT NULL);

END reports;

/

CREATE OR REPLACE PACKAGE BODY reports

AS

PROCEDURE print\_and\_send\_ps

(

i\_email\_addr IN VARCHAR2 DEFAULT NULL

)

IS

CURSOR cur\_read\_ps\_db IS

SELECT prob\_src\_nm

,prob\_key

,prob\_key\_txt

,prob\_notes

,sol\_notes

,seq

FROM (SELECT ps.prob\_src\_id

,ps.prob\_src\_nm

,p.prob\_key

,p.prob\_key\_txt

,p.prob\_notes

,ROW\_NUMBER() OVER(PARTITION BY s.prob\_id ORDER BY s.sol\_id) AS seq

,s.sol\_notes

FROM ps\_prob p

JOIN ps\_prob\_src ps

ON ps.prob\_src\_id = p.prob\_src\_id

JOIN ps\_sol s

ON s.prob\_id = p.prob\_id)

ORDER BY prob\_src\_id

,prob\_key

,seq;

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;

BEGIN

excp.assert(i\_email\_addr IS NOT NULL,'Destination email address required.', 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 cur\_read\_ps\_db LOOP

DECLARE

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);

EXCEPTION

WHEN OTHERS THEN *-- I hate this, but use here for brevity of demo*

logs.err;

END handle\_line;

BEGIN

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;

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.');

*-- Code used to email file here, but not possible on XE since Java engine not included*

END print\_and\_send\_ps;

END reports;

/

Conclusion

Looking at the sample code above, it is hoped the reader can see how easy it is to take comments and wrap them in logs.dbg() calls, then add a little context to provide runtime perspective to the debug messages, a little tagging to enable easy troubleshooting, and at least two calls to logs.info() at the start and end to record timestamps and elapsed times. It didn’t take more than a few minutes to fully instrument this otherwise bare code. If anything goes wrong with this process in the future, it could literally save hours to days of headscratching, downtime, poor-man’s debugging in lower databases where the problem can’t be replicated…you know, the usual.

Knowing that it takes only a couple minutes to download and install one of the open-source/free frameworks on the market, and start using it, it is my fondest wish the reader is inspired to do so now, today and not postpone any further the heavenly state that is instrumented code.

1. Or manually issuing ALTER PACKAGE <pkg\_name> COMPILE DEBUG PACKAGE | BODY [↑](#footnote-ref-1)
2. This is known as end-to-end identification. Oracle docs call it end-to-end metrics. I’ve given an entire presentation on this subject, so won’t go into detail here. [↑](#footnote-ref-2)
3. Note that the time\_remaining value in the v$session\_longops view should not be construed as 100% accurate. There are a number of variable that affect the accuracy of metrics in this view. One of them is recursive SQL statements (like index updates and such) which don’t figure into the time remaining. [↑](#footnote-ref-3)
4. If modifying your application’s connection classes to pass the client\_id, also modify them to clear package state, application contexts, and session tags before returning the database connection to the pool. [↑](#footnote-ref-4)
5. The first parameter to ksdwrt must be 2 if writing to the alert log. [↑](#footnote-ref-5)
6. If auditing is needed only for a few tables, modify the script’s cursor’s WHERE before running it. [↑](#footnote-ref-6)
7. Despite a personal aversion to the plural form of entity names, the singular “LOG” could not be used as it was an Oracle keyword [↑](#footnote-ref-7)
8. Use PRAGMA EXCEPTION\_INIT to bind the local exception to the expected Oracle error number. [↑](#footnote-ref-8)