Oracle® Applications

Supportability Guide

Release 11*i*

Part No. B13548-02

July 2006

ORACLE

Oracle Applications Supportability Guide, Release 11*i*

Part No. B13548-02

Primary Author: Mildred Wang

Contributing Author: Kunal Kapur, Sandeep Khemani, Mike Xu

Contributor: Jim Benge, George Buzsaki, Michele Casalgrandi, Pranab Pradhan, Sowmya Subramanian, Susan Stratton, Suchi Upadhyayula

Contents

3 Diagnostic Security

4 Diagnostics Result Reporting

5 Launching Oracle Diagnostics

Send Us Your Comments

Oracle Applications Supportability Guide, Release 11*i*

Part No. B13548-02

Oracle welcomes customers' comments and suggestions on the quality and usefulness of this document. Your feedback is important, and helps us to best meet your needs as a user of our products. For example:

- Are the implementation steps correct and complete?
- Did you understand the context of the procedures?
- Did you find any errors in the information?
- Does the structure of the information help you with your tasks?
- Do you need different information or graphics? If so, where, and in what format?
- Are the examples correct? Do you need more examples?

If you find any errors or have any other suggestions for improvement, then please tell us your name, the name of the company who has licensed our products, the title and part number of the documentation and the chapter, section, and page number (if available).

Note: Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the new Applications Release Online Documentation CD available on Oracle MetaLink and www.oracle.com. It contains the most current Documentation Library plus all documents revised or released recently.

Send your comments to us using the electronic mail address: appsdoc_us@oracle.com

Please give your name, address, electronic mail address, and telephone number (optional).

If you need assistance with Oracle software, then please contact your support representative or Oracle Support Services.

If you require training or instruction in using Oracle software, then please contact your Oracle local office and inquire about our Oracle University offerings. A list of Oracle offices is available on our Web site at www.oracle.com.

Preface

Intended Audience

Welcome to Release 11*i* of the *Oracle Applications Supportability Guide*.

This guide assumes you have a working knowledge of the principles and customary practices of your business area. If you have never used Oracle Applications we suggest you attend one or more of the Oracle Applications System Administration training classes available through Oracle University. (See Other Information Sources for more information about Oracle training.)

This guide also assumes you are familiar with the Oracle Applications graphical user interface. To learn more about the Oracle Applications graphical user interface, read the *Oracle Applications User's Guide*.

See Other Information Sources for more information about Oracle Applications product information.

See Related Information Sources on page x for more Oracle Applications product information.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/ .

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

Related Information Sources

You can choose from many sources of information, including online documentation, training, and support services to increase your knowledge and understanding of Oracle Applications system administration.

If this guide refers you to other Oracle Applications documentation, use only the Release 11i versions of those guides.

Online Documentation

All Oracle Applications documentation is available online (HTML or PDF).

- **PDF Documentation** - See the Oracle Applications Documentation Library CD for current PDF documentation for your product with each release. The Oracle Applications Documentation Library is also available on Oracle*MetaLink* and is updated frequently.
- **Online Help** - Online help patches (HTML) are available on Oracle*MetaLink*.
- **About Documents** - Refer to the About document for the mini-pack or family pack that you have installed to learn about feature updates, installation information, and new documentation or documentation patches that you can download. About documents are available on Oracle*MetaLink*.

Related Guides

You can read the guides online by choosing Library from the expandable menu on your HTML help window, by reading from the Oracle Applications Documentation Library CD included in your media pack, or by using a Web browser with a URL that your system administrator provides.

If you require printed guides, you can purchase them from the Oracle Store at http://oraclestore.oracle.com.

Guides Related to All Products

Oracle Applications User's Guide

This guide explains how to enter data, query, run reports, and navigate using the graphical user interface (GUI) available with this release of Oracle Advanced Product Catalog (and any other Oracle Applications products). This guide also includes information on setting user profiles, as well as running and reviewing reports and concurrent processes.

You can access this user's guide online by choosing "Getting Started with Oracle Applications" from any Oracle Applications help file.

Installation and System Administration

Oracle Applications Concepts

This guide provides an introduction to the concepts, features, technology stack, architecture, and terminology for Oracle Applications Release 11*i*. It provides a useful first book to read before an installation of Oracle Applications. This guide also introduces the concepts behind Applications-wide features such as Business Intelligence (BIS), languages and character sets, and Self-Service Web Applications.

Installing Oracle Applications

This guide provides instructions for managing the installation of Oracle Applications products. In Release 11*i*, much of the installation process is handled using Oracle Rapid Install, which minimizes the time to install Oracle Applications, the Oracle8 technology stack, and the Oracle8*i* Server technology stack by automating many of the required steps. This guide contains instructions for using Oracle Rapid Install and lists the tasks you need to perform to finish your installation. You should use this guide in conjunction with individual product user guides and implementation guides.

Upgrading Oracle Applications

Refer to this guide if you are upgrading your Oracle Applications Release 10.7 or Release 11.0 products to Release 11*i*. This guide describes the upgrade process and lists database and product-specific upgrade tasks. You must be either at Release 10.7 (NCA, SmartClient, or character mode) or Release 11.0, to upgrade to Release 11*i*. You cannot upgrade to Release 11*i* directly from releases prior to 10.7.

Maintaining Oracle Applications

Use this guide to help you run the various AD utilities, such as AutoUpgrade, Auto Patch, AD Administration, AD Controller, AD Relink, License Manager, and others. It contains how-to steps, screenshots, and other information that you need to run the AD utilities. This guide also provides information on maintaining the Oracle Applications file system and database.

Oracle Applications System Administrator's Guide

This guide provides planning and reference information for the Oracle Applications System Administrator. It contains information on how to define security, customize menus and online help, and manage concurrent processing.

Oracle Alert User's Guide

This guide explains how to define periodic and event alerts to monitor the status of your Oracle Applications data.

Oracle Applications Developer's Guide

This guide contains the coding standards followed by the Oracle Applications development staff. It describes the Oracle Application Object Library components needed to implement the Oracle Applications user interface described in the *Oracle Applications User Interface Standards for Forms-Based Products*. It also provides information to help you build your custom Oracle Forms Developer forms so that they integrate with Oracle Applications.

Oracle Applications User Interface Standards for Forms-Based Products

This guide contains the user interface (UI) standards followed by the Oracle Applications development staff. It describes the UI for the Oracle Applications products and how to apply this UI to the design of an application built by using Oracle Forms.

Other Implementation Documentation

Oracle Applications Product Update Notes

Use this guide as a reference for upgrading an installation of Oracle Applications. It provides a history of the changes to individual Oracle Applications products between Release 11.0 and Release 11*i*. It includes new features, enhancements, and changes made to database objects, profile options, and seed data for this interval.

Multiple Reporting Currencies in Oracle Applications

If you use the Multiple Reporting Currencies feature to record transactions in more than one currency, use this manual before implementing Oracle Applications. This manual details additional steps and setup considerations for implementing Oracle Applications with this feature.

Multiple Organizations in Oracle Applications

This guide describes how to set up and use Oracle Applications' Multiple Organization support feature, so you can define and support different organization structures when running a single installation of Oracle Applications.

Oracle Workflow Administrator's Guide

This guide explains how to complete the setup steps necessary for any Oracle Applications product that includes workflow-enabled processes, as well as how to monitor the progress of runtime workflow processes.

Oracle Workflow Developer's Guide

This guide explains how to define new workflow business processes and customize existing Oracle Applications-embedded workflow processes. It also describes how to define and customize business events and event subscriptions.

Oracle Workflow User's Guide

This guide describes how Oracle Applications users can view and respond to workflow notifications and monitor the progress of their workflow processes.

Oracle Workflow API Reference

This guide describes the APIs provided for developers and administrators to access Oracle Workflow.

Oracle eTechnical Reference Manuals

Each eTechnical Reference Manual (eTRM) contains database diagrams and a detailed description of database tables, forms, reports, and programs for a specific Oracle

Applications product. This information helps you convert data from your existing applications, integrate Oracle Applications data with non-Oracle applications, and write custom reports for Oracle Applications products. Oracle eTRM is available on Oracle*MetaLink*.

Training and Support

Training

Oracle offers a complete set of training courses to help you and your staff master Oracle Applications and reach full productivity quickly. These courses are organized into functional learning paths, so you take only those courses appropriate to your job or area of responsibility.

You have a choice of educational environments. You can attend courses offered by Oracle University at any one of our many Education Centers, you can arrange for our trainers to teach at your facility, or you can use Oracle Learning Network (OLN), Oracle University's online education utility. In addition, Oracle training professionals can tailor standard courses or develop custom courses to meet your needs. For example, you may want to use your organization's structure, terminology, and data as examples in a customized training session delivered at your own facility.

Support

From on-site support to central support, our team of experienced professionals provides the help and information you need to keep Oracle Applications working for you. This team includes your Technical Representative, Account Manager, and Oracle's large staff of consultants and support specialists with expertise in your business area, managing an Oracle Database, and your hardware and software environment.

Do Not Use Database Tools to Modify Oracle Applications Data

Oracle STRONGLY RECOMMENDS that you never use SQL*Plus, Oracle Data Browser, database triggers, or any other tool to modify Oracle Applications data unless otherwise instructed.

Oracle provides powerful tools you can use to create, store, change, retrieve, and maintain information in an Oracle database. But if you use Oracle tools such as SQL*Plus to modify Oracle Applications data, you risk destroying the integrity of your data and you lose the ability to audit changes to your data.

Because Oracle Applications tables are interrelated, any change you make using an Oracle Applications form can update many tables at once. But when you modify Oracle Applications data using anything other than Oracle Applications, you may change a row in one table without making corresponding changes in related tables. If your tables get out of synchronization with each other, you risk retrieving erroneous information and you risk unpredictable results throughout Oracle Applications.

When you use Oracle Applications to modify your data, Oracle Applications automatically checks that your changes are valid. Oracle Applications also keeps track of who changes information. If you enter information into database tables using database tools, you may store invalid information. You also lose the ability to track who has changed your information because SQL*Plus and other database tools do not keep a record of changes.

1

Oracle Diagnostics Overview

Introduction

Oracle Diagnostics improves the supportability of Oracle Applications by enabling the creation and execution of diagnostic tests. With Oracle Diagnostics, you can accomplish the following:

- Execute tests to prevent and troubleshoot problems.
- Be alerted automatically when problems occur.
- Find instructions to help you resolve problems on your own.
- Easily share detailed information about a problem with Oracle Support.
- Create your own tests to enhance the supportability of your system.

This chapter provides a high-level overview of the architecture, terminology, and features of Oracle Diagnostics.

Target Audiences

The target audiences for this manual are as follows:

System Administrators

As a system administrator, running diagnostic tests allows you to check the health of your system. You can use diagnostic tests to identify and resolve problems related to:

- Environment
- Post-installation setup
- Customization
- Any other functional problems

If you cannot successfully resolve the problem on your own, you can send the information generated by the tests to Oracle Support.

Implementation Engineers

When performing implementations, you are advised to run diagnostics after every installation or patching process to confirm that the environment is set up correctly and there are no outstanding issues to be resolved.

Application Developers and Consultants

You can use this manual to learn how to extend the diagnostic tests provided by Oracle. By creating your own diagnostic tests, you can diagnose issues specific to your implementation and diagnose any customizations or extensions that may exist.

Terminology

Applications

Refers specifically to the applications in the **FND_APPLICATION** table.

Groups

An ordered set of one or more related test cases. Every application has one or more groups.

Diagnostic Test

A diagnostic test case can be written in Java, written in PL/SQL, or created declaratively. It checks the correct behavior of a particular feature or business function.

Prerequisites

Each group can have one or more groups as prerequisites. The existence of a prerequisite implies that the current group will not execute correctly if the prerequisite group has not executed successfully.

Basic Mode

There are two modes associated with diagnostic tests. When writing tests, a developer can set up a test to be executable in basic mode, advanced mode (see below), or both modes. In basic mode, a test is executed with no user interaction, because the test either requires no input values or has a pre-configured set of input values.

Advanced Mode

In advanced mode, users can supply specific input values for execution of a diagnostic test.

Architecture

Oracle Diagnostics provides a framework to integrate diagnostic test cases and automate the execution of these test cases. It also provides a mechanism to write and run unit test cases.

Oracle Diagnostics is robust enough to function even when the environment and application are not set up properly. A set of "SYSTEM_TESTS" makes sure that the basic environment and installation are up and running. Information about these test cases is stored in a system resource file that can be read even when the application database is down. Metadata about diagnostic tests is stored in diagnostic-specific tables in the application schema.

Oracle Diagnostics can be launched from Oracle Applications Manager as well as the CRM System Administrator Console. If for some reason Oracle Applications Manager and/or the CRM System Administrator Console is not working, you can launch

Oracle Diagnostics through the following page: **jtfqalgn.htm**. When launched in this manner, the Oracle Diagnostics will attempt a guest user login. If this does not succeed, it will run in **noSession** mode. In this mode, only tests under "HTML Platform" and tests with low sensitivity security levels can be executed. The purpose of this mode is to help troubleshoot scenarios when Oracle Applications are not available.

Supported Features

Oracle Diagnostics supports the following features:

- Diagnostic tests that can be written in either Java or PL/SQL.
- Diagnostic tests can be written declaratively (without coding) through the Diagnostics UI.
- A user-friendly Web-based user interface.
- Metadata about diagnostic tests that is stored in diagnostic schema:
  - Diagnostic tests are grouped together in logical groups. Test groups belong to a particular application.
  - Multiple sets of input values for diagnostic tests can be stored in the database and also shipped as preseeded data.
- Diagnostic test results are persisted in the database, and a report viewer is available through the Web-based UI for viewing historical reports.
- Each diagnostic test can be an individual test case, or behave as a container that is made up of a set of individual test cases (called dependencies).
- Java-based diagnostic tests can be pipelined to pass output parameters from one test to another test within a container test.
- Parameters can be defined to be secure. This ensures that the parameter values are never displayed in visible text to the user.
- Parameters can be associated with LOVs.
- Oracle Diagnostics has a built-in notion of security. It checks the responsibility list of the user and determines if any of the Diagnostic Roles are tied to the user's responsibilities. For details, see Diagnostic Security, page 3-1.
- Diagnostic Roles determine the set of operations that can be performed on test groups, based on the sensitivity of the test group. The Diagnostic Roles available are as follows:
  - Super User: For all test groups of all registered applications, a Super User can execute tests, perform configuration, view reports, and set up security.
  - Application Super User: For all test groups of a given application, an Application Super User can execute tests, perform configuration, view reports, and set up security. For test groups of medium or low sensitivity belonging to other applications, an Application Super User can execute tests, configure test inputs, and view reports.
  - End User: For test groups of low sensitivity belonging to any application, an End User can execute tests and configure test inputs.
- Diagnostic test results can be e-mailed.

User Interfaces

Oracle Diagnostics has a Web-based user interface and a command-line user interface. You can:

- Execute tests, both basic and advanced.
- Register new applications.
- Register new groups.
- Modify or delete existing groups and their attributes such as group name, group sequence, and group prerequisites.
- Register new test cases in PL/SQL and Java.
- Modify or delete existing test cases.
- Set up one or more set of default values for a test.

Interfaces

The following is a list of the ways in which Oracle Diagnostics can be accessed. For further information, see Launching Oracle Diagnostics, page 5-1.

- http://<*domain_name*>/OA_HTML/jtfqalgn.htm
- http://<*domain_name*>/OA_HTML/jtflogin.jsp - This leads to the CRM System Administrator Console. After logging in, click the **Diagnostics** tab to launch the Oracle Diagnostics UI.
- Oracle Applications Manager - Diagnostics can be executed through Oracle Applications Manager (OAM) as of Release 11.5.9. After opening OAM to the Applications Dashboard, click the **Diagnostics** tab to see statistics and log data. The complete Oracle Diagnostics UI can be launched by clicking the **Launch Diagnostic Tests** button.

2

Developing Diagnostic Tests

Test Development Overview

This section describes how to develop diagnostic tests. Diagnostic tests can be written in Java or PL/SQL. Regardless of the test type, designing a useful test case should include the following steps:

1. Determine what to diagnose.
2. Determine what information is needed to execute the test (the input parameters).
3. Determine the core code paths to test.
4. Determine the different error cases that this test can expose.
5. Determine how to resolve the error scenarios (fix information).
6. Implement the test.
7. For security purposes, determine the test sensitivity level. For details, see Diagnostic Security, page 3-1.

The following sections describe the different types of test cases, and how to write test cases in Java and PL/SQL.

Diagnostic Test Categories

Ideally, each product or project should have diagnostic tests that cover each of the following categories:

Environmental Problems

This category includes tests that check whether or not a given version of a certain technology is present. For example, CRM Technology Foundation (JTT) checks the versions of the JDK and JDBC drivers. Similarly, if your product requires a specific workflow engine version, that test would belong in this category.

Installation Problems

This category includes tests that check if the installation has completed successfully. For example, you could write tests that check if the –D parameters that the application needs are configured correctly. You could also write tests that check if the servlets that the application needs have been installed.

Postinstallation and Setup Problems

This category includes tests regarding application post-installation steps. For example, you can test if the guest user has been set up properly.

Seed Data Issues

Seed data often becomes corrupted during patching. Tests in this category check that the application seed data is present and valid.

Customization Issues

If your application supports customer-site customizations, you should write tests to check the validity of all possible customizations.

Common Functional Issues

This category includes tests that address problems resulting from incorrect functional setups. For example, a task may not be assigned to a field service representative if the dispatch type is not set correctly. Thus you would write a test to check the dispatch type.

Developing Java Diagnostic Tests

This section describes how to develop diagnostic test cases in Java.

Preliminary Requirements for Java Tests

All Java diagnostic test cases must do the following:

- Extend oracle.apps.jtf.regress.qatool.QATestImpl.
- Have a no argument constructor.
- Call "super()" in the first line of the constructor.

Java Test Properties

Set up the following test properties as described. Typically they are set up in the test constructor.

- **String testName**

  The name of the test, which appears to the end user in the UI. It is inherited and should be set in the constructor.

- **String testDescription**

  Tells the end user the purpose of the test. It is inherited and should be set in the constructor.

- **String testedComponentName**

  The logical product component which this test is testing. It will be seen by end users. It is inherited and should be set in the constructor.

- **Integer mode**

  This property can be set to one of three values:

  - **QATestInterface.BASIC_MODE**

The test is run with minimal user interaction and as part of the group it belongs to. If the test requires inputs, then the values will be obtained from preconfigured values.

- **QATestInterface.ADV_MODE**

  The test can only be run individually. Typically tests that are used for probing the system for specific input values fall in this category. Inputs are inserted by the end user when the test is invoked.

- **QATestInterface.BOTH_MODE**

  The test can be executed in either basic or advanced mode. Most tests fall into this category.

- **version.setClass(QATestImpl test)**

  The version object is inherited and already instantiated. Calling this method on the object sets the RCS_ID of this test in the version object.

- **version.addClassName(String fullName)**

  This method sets up the version information of the core classes that the test diagnoses. Oracle Diagnostics implicitly adds the class names and their versions (RCS_ID) to the diagnostic report generated when a test is run. This information is useful when diagnosing problems on a customer instance.

- (Optional) **void addInput(QATestInput input)**

  Calling this API in the constructor adds a single input parameter to the test. Call this API once for each of the inputs that the test needs. Inputs can be of three types: **normal** (displayed as clear text), **secure** (not displayed as clear text), or **LOV** (value from a list of values). Parameters are wrapped by a QATestInput object. Details on how to implement an LOV input can be found in Integrating LOVs With Diagnostics, page 2-39.

  Note: For test cases that need to start a user session, see Diagnostic Security, page 3-1. This is important because using passwords as input parameters is highly discouraged.

- (Optional) **String[] dependentClassNames**

  Tests can specify child tests that are to be executed after this test. The children should also be well-formed Java diagnostic test cases. This variable can be set to a String array of the fully qualified class names of the children.

- (Optional) **boolean isDependencyPipelined**

  If set to true, it specifies that this test and its dependencies should be run with chaining of input values. Most tests do not need this feature, thus this variable is set to false by default. For more details about pipelining dependencies, see Pipelining Dependencies, page 2-19.

- (Optional) **boolean needNewRequest()**

  The only test property that is not specified in the constructor. Instead, if the test needs to be executed with a new HttpServletRequest and HttpServletResponse object, then the test should overload this method and return true. Setting this to true implies an extra round trip between the server and the client. Tests that need to push cookies onto the client may need this feature. However, most tests do not, and do not have to overload the method.

Java Test Execution

The execution logic is written in one of two versions of the runTest method. Both must be implemented by the test case. Two signatures of the runTest method exist, in order to support execution of the test in the context of a servlet and in a standalone command-line mode. Most diagnostic tests are executed through the Web-based UI as a servlet. Returning true signals success; returning false signals failure. The logic written in the runTest methods is executed in its own thread. Tests that need a framework session should start one at the beginning of the runTest and should be sure to end the session before the runTest method returns.

- **boolean runTest()**

  Is called if the test case is executed in a non-JSP environment and therefore does not take in the HttpServletRequest and HttpServletResponse. Returns true for success, false for failure. For tests that do not require the request/response objects, implement all test logic here, and have the other runTest delegate the call to this runTest.

- **boolean runTest(HttpServletRequest request, HttpServletResponse response)**

  Is called if the test case is executed in a JSP environment. Returns true for success, false for failure.

- (Optional) **Object getInputValue(String inputName)**

  Is called at the beginning of the runTest method to retrieve input values seeded in the database (if in basic mode) or by the user (if in advanced mode). The object type returned is dependent on the input type:

  - **String** if the input was type String, Secure String, or LOV.
  - **Integer** if the input was type Integer.

Java Test Reporting

The following reporting APIs are called after the runTest has completed.

- **String getReport()**

  Should give detailed information about the test execution, and will be displayed to the end user. For example, in the JTF Menu test, we render the given user's menu tree. This is displayed regardless of whether the test succeeds or fails. If HTML tags are involved in the formatting, then the returned string should begin with "@html ".

- **String getError()**

  If the test fails, then this should give the end user details about the failure and its cause. If HTML tags are involved in the formatting, then the returned string should begin with "@html ".

- **String getFixInfo()**

  If the test fails, then this should give information to the end user about how to resolve the error. If HTML tags are involved in the formatting, then the returned string should begin with "@html ".

- (Optional) **boolean isWarning()**

  If returns true, then tells framework to interpret success as status "Success with Warnings". The test result reflects this status to the end user, prompting them to view the report. This is not be called if the test is a failure.

- (Optional) **boolean isFatal()**

  If returns true, tells the framework to interpret a failure as a "Severe Error", in which case no other execution will follow this test.

- **(Optional) Hashtable getOutputValues()**

  If this test is part of a dependency pipeline, then this will be called to get String name-value pairs to be passed as inputs to the next test. These pairs are added to initial inputs, and inputs having the same name are overridden.

You can use **oracle.apps.jtf.regress.qatool.testcase.SampleTest** as a template to develop your test cases.

Java Diagnostic Test Sample Code

```
package oracle.apps.jtf.regress.qatool.testcase;

/* These two imports are neccessary **/
import oracle.apps.jtf.regress.qatool.*;
import javax.servlet.http.*;

import java.util.*;

public class SampleTest extends QATestImpl {

  /* Standard RCS_ID needed by all files */
  public static final String RCS_ID = "$Header$";

  public SampleTest() {
    super();  /* call the default constructor */

    /* Set test information */
    testName = "Sample";
    testDesc = "A template for developers to use when writing test
 cases";
    componentName = "Diagnostic Framework";
    mode = QATestInterface.BOTH_MODE; //or BASIC_MODE, ADVANCED_MO
DE

    /* Set version information. */
    version.setClass(this); // Set the version of myself

    /* Add version information of all the application classes you
are testing */
    version.addClassName("oracle.apps.jtf.menu.Menu");
    version.addClassName("oracle.apps.jtf.region.Region");

    /* Dependency Classes if any needs to be set up */
    /* If this test is made up of other test classes, then add dep
endent class names here */
    dependentClassNames = dependencies;


    /* Indicate if dependent classes should be pipelined */
    /* i.e., outputs from one test goes as inputs into the next te
st */
    isDependencyPipelined = true;
```

```
     /* Define parameter list (if any) used by runTest(..) method
*/
     /* End users will be able to set up values for this parameter
 list through the Admin UI */
     /* You can specify if the input is secure - i.e., should not
be displayed in clear text */
      addInput(new QATestInput("username", "SYSADMIN"));
 // default: not secure
      addInput(new QATestInput("resp ID", new RespLovImpl(), "2184
1")); //lov enabled input.  See the "Lov Integration with Diagnost
ics" section for more detailed information.
}
  /*** There are methods you HAVE to implement ***/

  public boolean runTest()
{
    / * This method is used when tests run through command-line.
      * If this test is only run in JSP mode, then simply return f
alse */
    return false;  // Fail!
  }

  public boolean runTest(HttpServletRequest request, HttpServletRe
sponse response)
{
    /* this method is used when tests run through JSP mode.
     * Implement what the test actually does */
}

 /**
   * Return the error that runTest(..) encountered
   **/
  public String getError() {
    return "TestException: Sample Test failed: Missing -D paramete
r ";
  }

/**
   * Return the fix (if any) for this error
   **/
  public String getFixInfo() {
    return "Make sure you pass the -D parameter to the Jserv";
  }

/**
 * Should other tests be run, given that this test failed!
**/
  public boolean isFatal() {
    return false; // not fatal
  }

  /* If there are dependent classes, state them here */
  private static final String[] dependencies =
                {"oracle.apps.jtf.regress.qatool.testcase.SetCook
ieTest",
                 "oracle.apps.jtf.regress.qatool.testcase.GetCook
ieTest"};
}
```

Report Formatting Library

A report formatting library is available to teams who want to:

- Simplify report generation.
- Have a consistent look and feel across different test reports.
- Intelligently generate HTML or text based reports.

The test case must instantiate a Report object using the createReportFormatter API that is defined in the QATestImpl.java base class. This method will return an object that implements the Report interface. Depending on the context of the test execution, this object will generate a report in either HTML or text.

The test case constructs the test report by populating the Report object with the correct contents. The following are descriptions of selected key methods, to demonstrate how the Report object is used.

Example Methods

beginSection(String sectionName, String sectionDescription)

For a top-level section (that is, one that has no non-ended beginSections prior to it), this will render a quicklink at the top of the report for easy navigation. All other formatting commands will be indented under the section until the section is ended. It is possible to create sub-sections by nesting beginSections. However, quick-links will not be generated for sub-sections.

endSection()

Ends the last beginSection.

printError(String errorMessage, String fixInformation)

Adds an error message to the report, along with a message on how the customer can resolve the error. If this error message occurs within a section, then the quick-link will be rendered to signal that an error occurred within the section.

printWarning(String errorMessage, String fixInformation)

Behaves like printError, except it will signal to the user that it is only a warning. The section's quicklink will be rendered as in printError.

println(String output)

Adds a string to the section with a new line. Methods are also available to print Java primitives.

printTable(String title, String summary, String[] headers, String[][] values)

Renders a table of information. Column titles are supplied by "headers" and table values are specified by "values".

String formatNoteLink(String name, String ID)

Creates a link to an Oracle*MetaLink* note, where "name" is the displayed link name and "ID" is the Oracle*MetaLink* note number. Unlike the previous formatting APIs, this is not automatically added to the report, but is instead returned as a formatted String. The String can be added to the report by calling println(…), and so on.

String getReportContents()

This should be called after Report object has been populated. It returns a string with the formatted report contents and should be returned in the getReport() method.

Sample Code

```
import oracle.apps.jtf.regress.qatool.report.Report;
…

public class MyReportTest extends QATestImpl {

    Report report = null;   //instance variable

    public boolean runTest() {
    report = createReportFormatter();
    report.beginSection("Profile Setup", "Validate if profiles …")

        // test code and report construction
        . . .
    }

    . . .

    public String getReport() {
        if (report == null) {
        return "";
        } else {
            return report.getReportContents();
        }
    }

    . . .
}
```

Report Interface

```
package oracle.apps.jtf.regress.qatool.report;

import java.io.*;

/**
 *  Diagnostic Report. This class handles all the formatting of th
e diagnostic
 *  report, and enforces output (look and feel) consistency and st
andards
 *
 */
public interface Report
{
    public static final String RCS_ID = "$Header: Report.java $";
    /*
     *  Methods to support QATestInterface required methods
     */
    /**
     *  Returns <code>true</code> if errors exist in the report
     *
     * @return    <code>true</code> if errors exist
     */
    public boolean getExceptionsExist();


    /**
```

```
     *  Returns <code>true</code> if errors exist in the report
     *
     * @return    <code>true</code> if errors exist
     */
    public boolean getErrorsExist();


    /**
     *  Returns <code>true</code> if warnings exist in the report
     *
     * @return    <code>true</code> if warnings exist
     */
    public boolean getWarningsExist();


    /**
     *  Returns the full report (including formatting).
     *
     * @return    The reportContents value
     */
    public String getReportContents();


    /*
     *  Implementor usable methods (non-formatting)
     */
    /**
     *  Sets the footer to be printed at the bottom of the report.
     *
     *  @param footer The footer
     */
    public void setFooter(String footer);
    /*
     *  Formatting methods
     */
    /**
     *  Gets the current indent level
     *
     * @return    The current indent level
     */
    public int getIndentLevel();


    /**
     *  Starts a new section in the report. If this is a top level
 section,
     *  the <code>sectionName</code> will be printed (in bold for
HTML)
     *  and a quicklink is automatically added. The output after i
t
     *  (i.e. the section contents) will be indented
     *  appropriately.
     *
     * @param  sectionName  Section name to print
     */
    public void beginSection(String sectionName);
```

```
    /**
     *  Starts a new section in the report. If this is a top level
 section,
     *  the <code>sectionName</code> will be printed (in bold for
HTML). The
     *  output after it (i.e. the section contents) will be indent
ed
     *  appropriately.
     *
     * @param  sectionName  Section name to print
     * @param  quickLink    Add this section to quicklinks
     */
    public void beginSection(String sectionName, boolean quickLink
);


    /**
     *  Starts a new section in the report. If this is a top level
 section,
     *  the <code>sectionName</code> will be printed (in bold for
HTML)
     *  and a quicklink is automatically added. The
     *  output after it (i.e. the section contents) will be indent
ed
     *  appropriately.
     *
     * @param  sectionName  Section name to print
     * @param  sectionDesc  Section description to print
     */
    public void beginSection(String sectionName, String sectionDes
c);


    /**
     *  Starts a new section in the report. If this is a top level
 section,
     *  the <code>sectionName</code> will be printed (in bold for
HTML). The
     *  output after it (i.e. the section contents) will be indent
ed
     *  appropriately.
     *
     * @param  sectionName  Section name to print
     * @param  sectionDesc  Section description to print
     * @param  quickLink    Add this section to quicklinks
     */
    public void beginSection(String sectionName, String sectionDes
c, boolean quickLink);


    /**
     *  Ends a section (and outdents any following output).
     */
    public void endSection();


    /**
     *  Add a quicklink to the current location in the report. Doe
```

```
sn't print
     *  any visible text to the current location in the report.
     *
     * @param  name  Quicklink name to show in TOC
     */
    public void addQuickLink(String name);


    /**
     *  Adds the error message and fix information to the report.
<code>ERROR</code>
     *  and <code>ACTION</code> text and formatting are added auto
matically.
     *
     * @param  errorMessage    Error message
     * @param  fixInformation  Fix information (ACTION)
     */
    public void printError(final String errorMessage, final String
 fixInformation);


    /**
     *  Adds the exception message and fix information to the repo
rt. <code>ERROR</code>
     *  and <code>ACTION</code> text and formatting are added auto
matically.
     *
     * @param  t        Throwable to print
     * @param  message  Description of where the message was caugh
t
     */
    public void printException(Throwable t, String message);


    /**
     *  Adds the warning message and fix information to the report
. <code>WARNING</code>
     *  and <code>ACTION</code> text and formatting are added auto
matically.
     *
     * @param  warningMessage  Warning message to print
     * @param  fixInformation  Fix information (ACTION)
     */
    public void printWarning(final String warningMessage, final St
ring fixInformation);


    /**
     *  Adds the notice message to the report. <code>ATTENTION</co
de> text and
     *  formatting are added automatically.
     *
     * @param  noticeMessage  Notice message to print
     */
    public void printNotice(final String noticeMessage);


    /**
```

```
     *  Adds the notice message and fix information to the report.
 <code>ATTENTION</code>
     *  and <code>ACTION</code> text and formatting are added auto
matically.
     *
     * @param  noticeMessage   Warning message to print
     * @param  fixInformation  Fix information (ACTION)
     * @deprecated             Diagnostic standards do not allow f
or an action
     *      to be specified for Notice/Attention messages
     */
    public void printNotice(String noticeMessage, String fixInform
ation);


    /**
     *  Add a blank line
     */
    public void println();


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  Text to print
     */
    public void println(String output);


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  Object to print (calls output.toString)
     */
    public void println(Object output);


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  boolean to print
     */
    public void println(boolean output);


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  Character to print
     */
```

```
    public void println(char output);


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  Double to print
     */
    public void println(double output);


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  Float to print
     */
    public void println(float output);


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  integer to print
     */
    public void println(int output);


    /**
     *  Adds the output to the report. Line is terminated with cr/
lf (or
     *  &gt;br&lt; for html)
     *
     * @param  output  Long to print
     */
    public void println(long output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
     *  html)
     *
     * @param  output  Text to print
     */
    public void print(String output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
     *  html)
     *
```

```
     * @param  output  Object to print (calls output.toString)
     */
    public void print(Object output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
     *  html)
     *
     * @param  output  boolean to print
     */
    public void print(boolean output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
     *  html)
     *
     * @param  output  Character to print
     */
    public void print(char output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
     *  html)
     *
     * @param  output  Double to print
     */
    public void print(double output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
     *  html)
     *
     * @param  output  Float to print
     */
    public void print(float output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
     *  html)
     *
     * @param  output  integer to print
     */
    public void print(int output);


    /**
     *  Adds the output to the report. Does not add cr/lf (or &gt;
br&lt for
```

```
     *  html)
     *
     * @param  output  Long to print
     */
    public void print(long output);


    /*
     * Table/Tree-Table related methods
     */
    /**
     *  Prints a table title and column headers. printTableRow sho
uld be
     *  called to add rows, and printTableClose must be called to
add the
     *  closing tags for the table.
     *
     * @param  title    Table title
     * @param  summary  Table summary (508)
     * @param  headers  Column headers
     */
    public void printTableHeader(String title, String summary, Str
ing[] headers);


    /**
     *  Prints a table title and column headers.printTableRow shou
ld be called
     *  to add rows, and printTableClose must be called to add the
 closing
     *  tags for the table.
     *
     * @param  summary  Table summary (508)
     * @param  headers  Column headers
     */
    public void printTableHeader(String summary, String[] headers)
;


    /**
     *  Prints a row in a table
     *
     * @param  cells  Cell values
     */
    public void printTableRow(String[] cells);


    /**
     *  Prints a formatted row in a table. Cell values are formatt
ed depending
     *  on the type of value.
     *
     * @param  cells  Cell values
     */
    public void printTableRow(Object[] cells);


    /**
```

```
     *  Prints a table row with the first column indented based on
 <code>level</code>
     *  . Used for printing heirarchy trees.
     *
     * @param  cells  Cell values
     * @param  level  Level in heirarchy (indent level)
     */
    public void printTreeRow(String[] cells, int level);


    /**
     *  Prints a table row with the first column indented based on
 <code>level</code>
     *  . Used for printing heirarchy trees.Cell values are format
ted
     *  depending on the type of value.
     *
     * @param  cells  Cell values
     * @param  level  Level in heirarchy (indent level)
     */
    public void printTreeRow(Object[] cells, int level);


    /**
     *  Adds closing tags for a table. Must be called before begin
ning any
     *  non-table output.
     */
    public void printTableClose();


    /**
     *  Adds closing tags for a table. Must be called before begin
ning any
     *  non-table output. String footer Table footer
     *
     * @param  footer  Table footer
     */
    public void printTableClose(String footer);


    /**
     *  Prints supplied values as table (in HTML as appropriate)
     *
     * @param  summary  Table summary (508 requirement)
     * @param  headers  Table column headings
     * @param  values   Table values
     */
    public void printTable(String summary, String[] headers, Strin
g[][] values);


    /**
     *  Prints supplied values as table (in HTML as appropriate)
     *
     * @param  title    Table title/caption
     * @param  summary  Table summary (508 requirement)
     * @param  headers  Table column headings
```

```
     * @param  values   Table values
     */
    public void printTable(String title, String summary, String[]
headers, String[][] values);


    /**
     *  Prints supplied values as table (in HTML as appropriate)
     *
     * @param  title    Table title/caption
     * @param  summary  Table summary (508 requirement)
     * @param  footer   Table footer
     * @param  headers  Table column headings
     * @param  values   Table values
     */
    public void printTable(String title, String summary, String fo
oter, String[] headers, String[][] values);


    /**
     *  Prints supplied values as table (in HTML as appropriate)
     *
     * @param  summary  Table summary (508 requirement)
     * @param  headers  Table column headings
     * @param  values   Table values
     * @param  levels   Indent level for the first column. This is
 the indent
     *      level, not the number of characters to indent.
     */
    public void printTree(String summary, String[] headers, String
[][] values, int[] levels);


    /**
     *  Prints supplied values as table (in HTML as appropriate)
     *
     * @param  title    Table title/caption
     * @param  summary  Table summary (508 requirement)
     * @param  headers  Table column headings
     * @param  values   Table values
     * @param  levels   Indent level for the first column. This is
 the indent
     *      level, not the number of characters to indent.
     */
    public void printTree(String title, String summary, String[] h
eaders, String[][] values, int[] levels);


    /**
     *  Prints supplied values as tree/table as appropriate for th
e output
     *  format
     *
     * @param  title    Table title/caption
     * @param  summary  Table summary (508 requirement)
     * @param  footer   Table footer
     * @param  headers  Table column headings
     * @param  values   Table values
```

```
     * @param  levels   Indent level for the first column. This is
 the indent
     *      level, not the number of characters to indent.
     */
    public void printTree(String title, String summary, String foo
ter, String[] headers, String[][] values, int[] levels);


    /*
     * Non-printing methods
     */
    /**
     *  Format a link to a Metalink note as appropriate for the ou
tput format
     *
     * @param  name  Document name
     * @param  id   Note number/Doc ID
     * @return       The link for printing (pass this to a printXX
X method)
     */
    public String formatNoteLink(String name, String id);


    /**
     *  Format a link to a Metalink note as appropriate for the ou
tput format
     *
     * @param  id  Note number/Doc id
     * @return     The link for printing (pass this to a printXXX
method)
     */
    public String formatNoteLink(String id);


    /**
     *  Format a link to a CR file as appropriate for the output f
ormat
     *
     * @param  name  File name/description
     * @param  id    CR File ID
     * @return       The link for printing (pass this to a printXX
X method)
     */
    public String formatCRLink(String name, String id);


    /**
     *  Format a link to a Metalink note as appropriate for the ou
tput format
     *
     * @param  name  Site name/description (ex. "Metalink")
     * @param  url   Site URL (ex. "http://metalink.oracle.com/")
     * @return       The link for printing (pass this to a printXX
X method)
     */
    public String formatLink(String name, String url);
```

```
    /*
     * Debug Methods
     */
    /**
     * Prints a list of report sections with the time consumed fro
m the
     * last section
     */
    public void printReportTiming();
}
```

Pipelining Dependencies

Certain tests can behave as container test cases, which contain a set of one or more tests to be executed in a specific order. The tests within a container test class are referred to as dependencies. When a test specifies dependencies to be run, it can also chain the dependencies so that the outputs of the previous tests are supplied as inputs for the next dependency in the pipeline. To do this, add the following line in the test constructor to tell the framework to pass outputs to the next test:

```
public SessionTest() {
...
dependentClassNames = new String[3];
dependentClassNames[0] =
  "oracle.apps.jtf.regress.qatool.testcase.authenticateTest";
dependentClassNames[1] =
  "oracle.apps.jtf.regress.qatool.testcase.createSessionTest";
dependentClassNames[2] =
  "oracle.apps.jtf.regress.qatool.testcase.ValidateLogoutSessionTe
st";

isDependencyPipelined = true;
}
```

Each test must specify the output values it wishes to pass to the next test. This is done by returning a Hashtable from the getOutputValues() API, which each test should implement if a member of a pipeline.

```
public Hashtable getOutputValues() {
        Hashtable out = new Hashtable();
        out.put("Username", this.username);
        out.put("<parameter-name>", "<parameter-value>");
        return out;
}
```

These outputs are not be passed to the next test unless that test specifies input parameters of the same name as the output parameters. For example, if authenticateTest did not specify a "Username" parameter in its constructor (via "addInput(...)"), then this value would not be passed to it.

Also, the input values that are specified by the main test and are given values from the database (in basic mode) or the user (in advanced mode) are automatically passed to each test specified as a dependency, whether the test is dependency-pipelined or not. However, if a test is dependency-pipelined, then it can override these original values before passing them to the next test.

Pipelining PL/SQL Scripts

First, you must write Diagnostics PL/SQL test scripts. There are three procedures that you should write specifically for supporting the pipelining dependency:

- `PROCEDURE getDependencies (package_names OUT NOCOPY JTF_DIAG_DEPENDTBL);`
- `PROCEDURE isDependencyPipelined (str OUT NOCOPY VARCHAR2);`
- `PROCEDURE getOutputValues(outputValues OUT NOCOPY JTF_DIAG_OUTPUTTBL);`

Samples of test code might look like:

```
PROCEDURE getDependencies (package_names OUT NOCOPY JTF_DIAG_DEPE
NDTBL) IS
BEGIN
 package_names := JTF_DIAGNOSTIC_ADAPTUTIL.initDependencyTable;

END getDependencies;

PROCEDURE isDependencyPipelined (str OUT NOCOPY VARCHAR2) IS
BEGIN
 str := 'FALSE';
END isDependencyPipelined;

PROCEDURE getOutputValues(outputValues OUT NOCOPY JTF_DIAG_OUTPUTT
BL) IS
 tempOutput JTF_DIAG_OUTPUTTBL;
BEGIN
 tempOutput := JTF_DIAGNOSTIC_ADAPTUTIL.initOutputTable;
 tempOutput := JTF_DIAGNOSTIC_ADAPTUTIL.addOutput(tempOutput,'Appl
ication ID', test_out);
 outputValues := tempOutput;
EXCEPTION
 when others then
 outputValues := JTF_DIAGNOSTIC_ADAPTUTIL.initOutputTable;
END getOutputValues;
```

Note that in the sample code above, in the procedure getOutputValues, the variable name 'test_out' should be defined in the specification file of the PL/SQL package. For example, define it as

```
test_out  VARCHAR2(200);
```

so that the 'test_out' variable can be assigned a value in other procedures like "runtest".

The following data types and methods support those three methods:

```
create or replace type JTF_DIAG_OUTPUTS as object  -- output valu
e entry
create or replace type JTF_DIAG_OUTPUTTBL as TABLE of JTF_DIAG_OUT
PUTS;  -- hashtable
create or replace type JTF_DIAG_DEPENDTBL as table of VARCHAR2(400
0);   -- dependencies

-- init/add output Hashtable
FUNCTION addOutput(outputs IN JTF_DIAG_OUTPUTTBL,var IN  VARCHAR2,
val IN  VARCHAR2) RETURN JTF_DIAG_OUTPUTTBL;
FUNCTION initOutputTable RETURN JTF_DIAG_OUTPUTTBL;

-- init/output dependency array
FUNCTION addDependency(dependencies IN JTF_DIAG_DEPENDTBL, val IN
 VARCHAR2) RETURN JTF_DIAG_DEPENDTBL;
FUNCTION initDependencyTable RETURN JTF_DIAG_DEPENDTBL;
```

Sample Master Test Case Containing Only Pipelined Test Names

Here is an example of a test case with only pipelined test names:

```
CREATE OR REPLACE PACKAGE BODY PIPE_DIAG_QAPACKAGE AS

  PROCEDURE getDependencies (package_names OUT NOCOPY JTF_DIAG_DEP
ENDTBL) IS
    tempDependencies JTF_DIAG_DEPENDTBL;
  BEGIN
    tempDependencies := JTF_DIAGNOSTIC_ADAPTUTIL.initDependencyTab
le;
    tempDependencies := JTF_DIAGNOSTIC_ADAPTUTIL.addDependency(tem
pDependencies,'INV_DIAG_QAPACKAGE');
    tempDependencies := JTF_DIAGNOSTIC_ADAPTUTIL.addDependency(tem
pDependencies,'OE_DIAG_QAPACKAGE');
    package_names := tempDependencies;
   EXCEPTION
   when others then
     package_names := JTF_DIAGNOSTIC_ADAPTUTIL.initDependencyTable
;
  END getDependencies;

  PROCEDURE isDependencyPipelined(str OUT NOCOPY VARCHAR2) IS
  BEGIN
    str := 'TRUE';
  END isDependencyPipelined;


  PROCEDURE getOutputValues(outputValues OUT NOCOPY JTF_DIAG_OUTPU
TTBL) IS
  BEGIN

   outputValues := JTF_DIAGNOSTIC_ADAPTUTIL.initOutputTable;
  END getOutputValues;

  PROCEDURE runtest(inputs IN  JTF_DIAG_INPUTTBL,
                            report OUT NOCOPY JTF_DIAG_REPORT,
                            reportClob OUT NOCOPY CLOB) IS
     statusStr   VARCHAR2(50);   -- SUCCESS or FAILURE
     errStr      VARCHAR2(4000); -- error message
```

```
     fixInfo     VARCHAR2(4000); -- fix tip
     isFatal     VARCHAR2(50);   -- TRUE or FALSE
   BEGIN
     JTF_DIAGNOSTIC_ADAPTUTIL.setUpVars;  -- must have

     -- html formatting
     JTF_DIAGNOSTIC_ADAPTUTIL.addStringToReport('@html');
     JTF_DIAGNOSTIC_COREAPI.Show_Header(null, null); -- add html c
ss

  -- NOTE: no any execution code needs to be put here!

     statusStr := '';
     errStr := '';
     fixInfo := '';
     isFatal := '';

     -- construct report
     report := JTF_DIAGNOSTIC_ADAPTUTIL.constructReport(statusStr,
errStr,fixInfo,isFatal);
     reportClob := JTF_DIAGNOSTIC_ADAPTUTIL.getReportClob;
   END runTest;

END PIPE_DIAG_QAPACKAGE;
```

Seamless Pipelining between Diagnostics Java and PL/SQL Scripts

Since PL/SQL pipelining and Java pipelining share the same procedure, you can pipeline the PL/SQL scripts and Java scripts as a bonus of implementing PL/SQL pipelining.

To pipeline the Java and PL/SQL test scripts, define the master test case, which only contains the test names of Java/PL/SQL test names, as either Java or PL/SQL test script. You can even pipeline Java scripts using a PL/SQL master script, and pipeline PL/SQL scripts using a Java master script.

Sample Master Script to Pipeline Java and PL/SQL Scripts

You can chose to write a master script in Java or PL/SQL.

The following is a sample PL/SQL script which uses the following method to drive the pipelining between a PL/SQL test case (INV_DIAG_QAPACKAGE3) and a Java test case (oracle.apps.jtf.regress.qatool.testcase.MenuTest).

```
PROCEDURE getDependencies (package_names OUT NOCOPY JTF_DIAG_DEPE
NDTBL) IS
    tempDependencies JTF_DIAG_DEPENDTBL;
  BEGIN
    tempDependencies := JTF_DIAGNOSTIC_ADAPTUTIL.initDependencyTab
le;
    tempDependencies := JTF_DIAGNOSTIC_ADAPTUTIL.addDependency(tem
pDependencies,'INV_DIAG_QAPACKAGE3');
    tempDependencies := JTF_DIAGNOSTIC_ADAPTUTIL.addDependency(tem
pDependencies,'oracle.apps.jtf.regress.qatool.testcase.MenuTest');

    package_names := tempDependencies;
   EXCEPTION
   when others then
     package_names := JTF_DIAGNOSTIC_ADAPTUTIL.initDependencyTable
;
  END getDependencies;
```

Developing PL/SQL Test Cases

If you are writing PL/SQL test cases, then use the following steps to successfully develop a new PL/SQL test case for Oracle Diagnostics.

1. Create a PL/SQL package under the **APPS** schema with a meaningful naming structure, like **<APP_ID>_<group_name>_<test_name>**. This to ensure that the package is immediately recognizable in the database and can be found when executing unit tests in the future.
2. Implement core APIs in the PL/SQL test package to plug into Oracle Diagnostics. These mandatory APIs must be declared in the package header section in order to be visible and accessible to the rest of the framework. The mandatory APIs are described in PL/SQL Package Test Case APIs, page 2-24. These procedures are briefly summarized below:
   - Implement a **runTest(..)** procedure to provide core test logic and write to PL/SQL out parameters for reporting results back to the framework.
   - Implement **getDefaultTestParams(..)** to provide the framework with test parameters (if needed).
   - Implement the **getTestName(..) , getComponentName(..),** and **getTestDesc(..)** procedures to feed test metadata back into the framework.
   - (Optional) Implement **init()** and **cleanup()** procedures to create/initialize and drop/free data structures or resources at the beginning and end of each test.
3. Follow package structure and guidelines:
   - All core APIs from Step 2 above must be declared in the package header in order to be accessible externally by Oracle Diagnostics.
   - Include the RCS version information for the package specification and body. This step is important because the RCS version information is used to determine the version of the test for reporting purposes.
4. Utilize the two helper packages which assist in the development of PL/SQL diagnostic tests:

- **JTF_DIAGNOSTIC_ADAPTUTIL**

  This package provides a set of procedures and functions for object initialization and manipulation of some of the PL/SQL data structures that are part of the framework. This package has a broad range of utility APIs. For example, one procedure retrieves an initialized CLOB, and another procedure that adds an input to another data structure. The complete API is described in the section PL/SQL Helper Packages, page A-1.

- **JTF_DIAGNOSTIC_COREAPI**

  This package provides a set of helper procedures and functions for standard formatting of reports. The package contains APIs that provide both HTML and plain text formatting. APIs are available to return formatted results from the database. The complete API is described in the section PL/SQL Helper Packages, page A-1.

PL/SQL Package Test Case APIs

Before developing a test package, you should familiarize yourself with the procedures that form the core of the PL/SQL diagnostics logic.

See also: Pipelining PL/SQL Scripts, page 2-20.

runTest

```
Procedure runTest (arg1 IN JTF_DIAG_INPUTTBL,
                   arg2 OUT JTF_DIAG_REPORT,
                   arg3 OUT CLOB)
```

This procedure is the main entry point for PL/SQL test execution. The test logic is executed within this procedure. During test execution, it propagates the **JTF_DIAG_REPORT** and **CLOB** objects with the test report messages and detailed report data respectively before being returned to the framework.

The **JTF_DIAG_INPUTTBL** object Is passed into the **runTest(..)** procedure from the framework. This object contains the input values for the test that were retrieved from the **getDefaultTestParams(..)** procedure. The **getDefaultTestParams(..)** is called by the framework prior to the **runTest(..)** procedure.

There are fourVARCHAR2 fields in a **JTF_DIAG_REPORT** object. The **status**, **errStr**, **fixInfo**, and **isFatal** fields, along with the report CLOB data, must be propagated in the event of an error occurring in the test. In such cases, the status string is set to **FAILURE**. For a successful test run, the status string is set to **SUCCESS**. On some occasions, the error which occurs is not sufficiently critical to halt execution. In this case, the status field should be set with the string **WARNING**. This setting has the effect of displaying the reportClob along with the error that occurred. All fields may populated as if an error had occurred, but test execution is not halted. If the test status is set to **SUCCESS**, then only the report CLOB needs to be written to with correct data. The report CLOB data (if any) is rendered to the UI by the framework irrespective of test failure, warning, or success.

- The **status** field that contains the result of the test is of type VARCHAR2. If the status field value is **SUCCESS**, then you are flagging that the test case passed. If the status field is **FAILURE**, then you are flagging a test case failure. If the status field is **WARNING**, then you are flagging a test case warning.

- The **errStr** field contains a string representation of the error (if any). This string can have up to 4000 characters. For example, you could set the error message to be a SQLERRM thrown by a caught exception.
- The **fixInfo** field contains a string providing suggestions on how to fix the error (if any). This string can have up to 4000 characters.
- The **isFatal** field contains a VARCHAR2 representation of a Boolean value. The isFatal value can either be "TRUE" or "FALSE". If the value is set to "TRUE", then the framework is informed that if the test has been reported as failing and that the current error is a fatal error.

These values can be assigned to their fields in the **JTF_DIAG_REPORT** object directly, or with a call to a procedure in the utility package, **JTF_DIAGNOSTIC_ADAPTUTIL.constructReport(..)**. This procedure takes the report fields described above and inserts them into the report object ready to be returned to the framework. The Diagnostic PL/SQL test case adapter massages the data coming back so that it is available to the framework's reporting and logging logic.

A **CLOB** object is returned to the framework upon completion of a test run, as it is also registered as an OUT parameter. This CLOB must be initialized before it can be used. You can do this with a call to the **JTF_DIAGNOSTIC_ADAPTUTIL.***setUpVars*function. Hence, we call this function immediately after commencing execution of the **runTest(..)** block. The CLOB object can be written to with calls to the **addStringToReport(..)** procedure call in the utility package **JTF_DIAGNOSTIC_ADAPTUTIL**. This call appends the passed-in line of text/string or LONG object to the overall report. Note that if the report is HTML-based, then the first string added to the report must be "@html".

Oracle Diagnostics provides a package called **JTF_DIAGNOSTIC_COREAPI**. It contains a library of APIs which provide for formatted HTML and plain text reporting. If you do not use this package, then you must provide all HTML formatting tags for the report (such as colors, new lines, and so on).

Calls to the core API package **JTF_DIAGNOSTIC_COREAPI** will write to the same report CLOB object. For example, the **JTF_DIAGNOSTIC_COREAPI.line_out(..)** procedure is the same as **JTF_DIAGNOSTIC_ADAPTUTIL.addStringToReport(..)** where both eventually write to the same CLOB.

All input variables can be retrieved with a call to the **getInputValue(..)** procedure in the utility package. This procedure passes in the name of the variable and returns the associated value. The value returned is of type VARCHAR2. You must convert this object to other types (INTEGER, NUMBER, etc.) if needed. Variable, value pairings must be made with calls to the **addInput(..)** procedure in the **getDefaultTestParams(..)** procedure.

The **runTest(..)** procedure returns a CLOB, which contains a detailed report for the framework. The PL/SQL OUT variable must reference the report CLOB when the **runTest(..)** procedure returns. The report CLOB can be retrieved with a call to **JTF_DIAGNOSTIC_ADAPTUTIL.getReportClob** and then be reassigned to the CLOB OUT variable. In light of this, each **runTest(..)** procedure typically has logic implemented before the test returns control to the framework. That is, at the end of a normal **runTest(..)** body block and in its exception handler.

Let's say that the test has OUT variables as below :

```
runTest(arg1  IN JTF_DIAG_INPUTTBL,
    arg2  OUT JTF_DIAG_REPORT,
     arg3  OUT CLOB)
```

Before returning to the framework, these OUT variables should be set and pointing to the correct **JTF_DIAG_REPORT** report and **CLOB**. The example below shows how this can be done:

```
arg2 := JTF_DIAGNOSTIC_ADAPTUTIL.constructReport('SUCCESS',
                                                 'Error occurred',
                                                     'Fix the Error',
                                                     'FALSE')
arg3 := JTF_DIAGNOSTIC_ADAPTUTIL.getReportClob;
```

The following code sample demonstrates a simple test which checks if the user name passed in has an account and is registered in the FND_USER table. It demonstrates how PL/SQL tests are written for Oracle Diagnostics and demonstrates some of the more important implementation details mentioned above.

```
1 PROCEDURE runtest(inputs IN  JTF_DIAG_INPUTTBL,
2                   report OUT JTF_DIAG_REPORT,
3                   reportClob OUT CLOB) IS
4  reportStr LONG;
5  counter   NUMBER;
6  c_userid  VARCHAR2(50);
7  statusStr VARCHAR2(50);
8  errStr    VARCHAR2(4000);
9  fixInfo   VARCHAR2(4000);
10 isFatal   VARCHAR2(50);
11 BEGIN
12   JTF_DIAGNOSTIC_ADAPTUTIL.setUpVars;
13   JTF_DIAGNOSTIC_ADAPTUTIL.addStringToReport('@html');
14   c_userid := JTF_DIAGNOSTIC_ADAPTUTIL.getInputValue('USERNAME'
,inputs);
15   SELECT COUNT(*) INTO counter
16   FROM  FND_USER
17   WHERE user_name  LIKE c_userid;
18   IF (counter > 0) THEN
19     reportStr := 'The test completed successfully';
20     JTF_DIAGNOSTIC_ADAPTUTIL.addStringToReport(reportStr);
21     statusStr := 'SUCCESS';
22   ELSE
23     statusStr := 'FAILURE';
24     errStr := 'This test failed as '||counter||' is less than 1
 ';
25     fixInfo := 'Make sure that the username entered is correct'
;
26     isFatal := 'FALSE';
27   END IF;
28   report :=  JTF_DIAGNOSTIC_ADAPTUTIL.constructReport(statusStr
,
                                                        errStr,
                                                        fixInfo,
                                                        isFatal);

29   reportClob := JTF_DIAGNOSTIC_ADAPTUTIL.getReportClob;
30   EXCEPTION WHEN others THEN
31     JTF_DIAGNOSTIC_COREAPI.errorprint('Error: '||sqlerrm);
32     JTF_DIAGNOSTIC_COREAPI.ActionErrorPrint('This is the except
ion
                                                handler');
33     statusStr := 'FAILURE';
34     errStr := sqlerrm ||' occurred in script - Exception handle
d';
35     fixInfo := 'Avoid throwing exceptions';
36     isFatal := 'FALSE';
37     report := JTF_DIAGNOSTIC_ADAPTUTIL.constructReport(statusSt
r,
                                                         errStr,
                                                         fixInfo,
                                                         isFatal)
;
38     reportClob := JTF_DIAGNOSTIC_ADAPTUTIL.getReportClob;
39   END runTest;
```

The following table provides descriptions of the lines in the above code sample.

Line Descriptions for Code Sample

Line(s)	Description
1-3	The runTest procedure takes the three arguments shown.
4-10	The declaration section of variables used in this example.
12	A call to initialize objects for the current session. Initializes a CLOB for report writing and initializes the global HTML formatting flag to false. Unless the first string in the report CLOB is "@html", this ensures that output is in plain text (the default setting).
13	If the report to be generated contains HTML formatting, then the first string written to the report must be "@html". In this case, HTML formatting has been turned on.
14	This line retrieves the value of the variable USERNAME and stores it locally in the **c_userid** variable. This is how the test retrieves inputs in the framework (variables are added in **getDefaultTestParams(..)** procedure).
15-17	A SQL query to demonstrate using the recently retrieved input value **c_userid**.
20	Writing a string to the report CLOB.
21	Setting a local variable with the SUCCESS string. This will be added to the outgoing JTF_DIAG_REPORT object later in the code when the constructReport(..) is called.
24	Constructing an error message to return.
25	Providing fix information if an error occurs.
26	Indicating with a VARCHAR2 object that if the error occurs, it is not a fatal error. Fatal errors have the ability to halt any following tests. True means fatal, false means not fatal.
28	Here the PL/SQL OUT parameter JTF_D IAG_REPORT is propagated with a call to the **constructReport(..)** procedure. The status string error description (if error occurs), fix Suggestion (if error occurs), and is error fatal (if error occurs) are added to the JTF_DIAG_REPORT object.
29	Here the PL/SQL OUT parameter CLOB is retrieved and gets assigned the report CLOB for the current session. The call **getReportClob()** retrieves the CLOB initialized by **setupVars** and written to with the **addStringToReport(..)** and **JTF_DIAGNOSTIC_COREAPI** procedural calls.

Line(s)	Description
31-32	Calling support APIs in the JTF_DIAGNOSTIC_COREAPI package.
33-36	See lines 21-26 above.
37-38	See lines 28 and 29 above. If an exception is being caught, then the report and reportClob objects have to be assigned and returned in the exception block. This behavior is similar to how they would be returned in the main block, lines 11-29.

getDefaultTestParams

```
Procedure getDefaultTestParams(arg1 OUT JTF_DIAG_INPUTTBL)
```

In this procedure, you should register any input parameters that the test needs, along with their default values. This procedure is executed separately from test execution in order to determine input values (if applicable). This procedure is called by the framework so that proper input fields are rendered through the framework UI. The framework requires all inputs that the test is to take as inputs. The **JTF_DIAG_INPUTTBL** object is propagated here with values and upon its return is then queried by the framework. Eventually this object is passed into the **runTest(..)** procedure while invoking test execution.

Calls to the utility package procedure **addInput(..)** add input variables to the **JTF_DIAG_INPUTTBL** object that is returned to the framework. The **addInput(..)** procedure is overloaded and by default displays the value field on the diagnostic UI. A call to the **addInput(..)** procedure with the **showValue** parameter set to "false" (Boolean value) hides the value field data on the UI. For example, if you want to add the parameter "USERNAME", then you might use something like the following:

```
1 PROCEDURE getDefaultTestParams(defaultInputValues OUT JTF_DIAG_
INPUTTBL)IS
2   tempInput JTF_DIAG_INPUTTBL;
3 BEGIN
4   tempInput := JTF_DIAGNOSTIC_ADAPTUTIL.initinputtable;
5   tempInput :=
       JTF_DIAGNOSTIC_ADAPTUTIL.addInput(tempInput,'USERNAME','SYS
ADMIN');
6   defaultInputValues := tempInput;
7 END getDefaultTestParams;
```

The following table provides descriptions of these code lines.

Line Description of Code Sample

Lines	Description
1	This method expects that **defaultInputValues** is passed in as an OUT parameter.
4	Create an initialized **JTF_DIAG_INPUTTBL** object called **temp** with a call to the **inputtable()** procedure.
5	Add a parameter called **USERNAME** with **SYSADMIN** as its default value. We pass in the **JTF_DIAG_INPUTTBL** to the **addInput(..)** procedure and add the parameter to it.
6	Assign the OUT variable to the object that contains all the recently added input variables.

getTestName

```
Procedure  getTestName(arg1 OUT VARCHAR2)
```

You should return the name of the test in this procedure. This procedure is accessed by the framework to query the name of the procedure. If the procedure is missing or throws an error, then the string "Unknown" will be returned in its place. The procedure returns a VARCHAR2 object that holds the name of the test. For example:

```
PROCEDURE getTestName(name OUT VARCHAR2) IS
BEGIN
  name := 'fnd_user User account test';
END getTestName;
```

getComponentName

```
Procedure getComponentName(arg1 OUT VARCHAR2)
```

You should return the name of the test component in this procedure. The framework accesses this procedure for the name of the component that this test case belongs to. If the procedure is missing or throws an error, then the string "Unknown" will be returned in its place. The procedure returns a VARCHAR2 object, which is the name of the test component. For example:

```
PROCEDURE getComponentName(name OUT VARCHAR2) IS
BEGIN
  name := 'User Account Tests';
END getComponentName;
```

getTestDec

```
Procedure getTestDesc(arg1 OUT VARCHAR2)
```

You should return a description of the test in this procedure. The framework accesses this procedure for the description of this test case. If the procedure is missing or throws an

error, then the String "No Description Available" will be used in its place. The procedure returns a VARCHAR2 object, which contains the description of the test. For example:

```
PROCEDURE getTestDesc(desc OUT VARCHAR2) IS
BEGIN
  desc :='fnd_user User account test-checks for a account in fnd_u
ser'
         'table';
END getTestDesc;
```

getTestMode

```
FUNCTION getTestMode RETURN INTEGER
```

This function returns the current test mode that the current PL/SQL test will operate as. This function is not mandatory and all tests will default to basic mode.

The mode returned by this function can be one of the following:

- JTF_DIAGNOSTIC_ADAPTUTIL.BASIC_MODE

  The test is run with minimal user interaction and as part of the group it belongs to. If the test requires inputs, then the values will be obtained from preconfigured values. This is the default mode.

- JTF_DIAGNOSTIC_ADAPTUTIL.ADVANCED_MODE

  The test can only be run individually. Typically tests that are used for probing the system for specific input values fall in this category. Inputs are inserted by the customer during test invocation.

- JTF_DIAGNOSTIC_ADAPTUTIL.BOTH_MODE

  The test can be executed in either basic or advanced mode. Most tests fall into this category.

An example of how a PL/SQL test would explicitly set itself to be an Advanced test follows below:

```
FUNCTION getTestMode return INTEGER IS
BEGIN
  return JTF_DIAGNOSTIC_ADAPTUTIL.ADVANCED_MODE;
END getTestMode;
```

init

```
Procedure init()
```

This procedure does not take any parameters and is always called prior to the **runTest** procedure being executed. In this procedure, implement the code for any data structures that need to be set up before the test runs. For example:

```
PROCEDURE init IS
BEGIN
  -- Example, to create a temporary table for the test
  -- execute immediate 'create table temp_qa(name VARCHAR2(30))';
    null;
END init;
```

cleanup

```
Procedure cleanup()
```

This procedure does not take any parameters and is called after the **runTest** procedure has been executed. In this procedure, implement the code for any data structures that need to be cleaned up after the test runs. Typically, these are the data structures that were set up in the **init()** call. You should still implement this procedure and include a null code block even if there is nothing to be done after the test is run. For example:

```
PROCEDURE cleanup IS
BEGIN
  -- Example, to drop the temporary table created in the init() ca
ll above.
  null;
END cleanup;
```

PL/SQL Utility Packages

As mentioned earlier, Oracle Diagnostics provides two helper packages for the PL/SQL diagnostic test writing process. For more information on these helper packages, see PL/SQL Helper Packages, page A-1. Note that the APIs exposed in the **JTF_DIAGNOSTIC_COREAPI** package are intended to facilitate the migration of test cases written by Oracle Support. Details about the migration support for scripts written by Oracle Support are also provided in the section PL/SQL Helper Packages, page A-1.

PL/SQL Diagnostic Test Sample Code

Below is a sample package of a diagnostic test case. It demonstrates areas the following test package criteria:

- HTML formatting is enabled with the "@html" string in the report CLOB.

  Be aware that calls to support APIs can still be made without the @html flag; the output will be in plain text.

- Calling support APIs in the **JTF_DIAGNOSTIC_COREAPI** package.

- Implementing the core diagnostic APIs.

```
CREATE OR REPLACE PACKAGE JTF_DIAG_FNDUSERACCOUNT AS
/* $Header: filename 115.xx YYYY/MM/DD 24:MM:SS userid [no]ship $
*/
  PROCEDURE init;
  PROCEDURE getDefaultTestParams(defaultInputValues OUT JTF_DIAG_I
NPUTTBL);
  PROCEDURE cleanup;
  PROCEDURE runtest(inputs IN  JTF_DIAG_INPUTTBL,
      report OUT JTF_DIAG_REPORT,
               reportClob OUT CLOB);
  PROCEDURE getComponentName(name OUT  VARCHAR2);
  PROCEDURE getTestName(name OUT  VARCHAR2);
  PROCEDURE getTestDesc(descStr OUT VARCHAR2);
END;
/

CREATE OR REPLACE PACKAGE BODY JTF_DIAG_FNDUSERACCOUNT AS
/* $Header: filename 115.xx YYYY/MM/DD 24:MM:SS userid [no]ship $
*/
------------------------------------------------------------
```

```
-- procedure to initialize test datastructures
-- executed prior to test run - leave body as null otherwize
-------------------------------------------------------------
PROCEDURE init IS
  BEGIN
-- test writer could insert special setup code here
   null;
  END init;

 -------------------------------------------------------------
 -- procedure to clean up any  test datastructures that were setup
 in the init
-- procedure call executes after test run - leave body as null oth
erwize
-------------------------------------------------------------
  PROCEDURE cleanup IS
  BEGIN
 -- test writer could insert special cleanup code here

   NULL;
  END cleanup;

-------------------------------------------------------------
-- procedure to execute the PLSQL test
-- the inputs needed for the test are passed in and a report objec
t and CLOB are -- returned.
-- note the way that support API writes to the report CLOB.
-------------------------------------------------------------
  PROCEDURE runtest(inputs IN  JTF_DIAG_INPUTTBL,
        report OUT JTF_DIAG_REPORT,
       reportClob OUT CLOB) IS
     reportStr LONG;
     counter   NUMBER;
     dummy_v2t JTF_DIAGNOSTIC_COREAPI.v2t;
     c_userid  VARCHAR2(50);
     statusStr VARCHAR2(50);
     errStr    VARCHAR2(4000);
     fixInfo   VARCHAR2(4000);
     isFatal   VARCHAR2(50);
     dummy_num NUMBER;
     sqltxt VARCHAR2 (2000);
   BEGIN
     JTF_DIAGNOSTIC_ADAPTUTIL.setUpVars;
     JTF_DIAGNOSTIC_ADAPTUTIL.addStringToReport('@html');
     JTF_DIAGNOSTIC_COREAPI.insert_style_sheet;
     JTF_DIAGNOSTIC_COREAPI.line_out('this also writes to the clob
');
       c_userid := JTF_DIAGNOSTIC_ADAPTUTIL.getInputValue('USERID'
,inputs);
     SELECT COUNT(*) INTO counter
     FROM  FND_USER
     WHERE user_name  LIKE c_userid;
     sqltxt := 'select segment1, attribute6 from pa_projects '||
          'where rownum < 5';
     dummy_num:= JTF_DIAGNOSTIC_COREAPI.display_sql(sqltxt,'Disp S
QL 2 params');
        IF (counter = 1) THEN
 reportStr := 'The test completed as expected the number of
```

```
                    accounts registered for '||c_userid||' in
                    fnd_user is '||counter;
       JTF_DIAGNOSTIC_ADAPTUTIL.addStringToReport(reportClob,repor
tStr);
       JTF_DIAGNOSTIC_ADAPTUTIL.addStringToReport('String into rep
ort');
       statusStr := 'SUCCESS';
     ELSE
       JTF_DIAGNOSTIC_COREAPI.ActionErrorPrint('You better do some
thing!');
       statusStr := 'FAILURE';
       errStr := 'This test failed as the accounts for the user '|
|c_userid||'
                 in fnd_user count " '||counter||' " is not = 1 '
;
                      fixInfo := 'Put informative fix info. here
';
       isFatal := 'FALSE';
                    END IF;
    report :=     JTF_DIAGNOSTIC_ADAPTUTIL.constructReport(statusS
tr,errStr,fixInfo,isFatal);
    reportClob := JTF_DIAGNOSTIC_ADAPTUTIL.getReportClob;
                 END runTest;

  ------------------------------------------------------------
  -- procedure to report name back to framework
  ------------------------------------------------------------
  PROCEDURE getComponentName(name OUT VARCHAR2) IS
  BEGIN
    name := 'SDF Migration tests';
  END getComponentName;

    ------------------------------------------------------------
  -- procedure to report test description back to framework
  ------------------------------------------------------------
  PROCEDURE getTestDesc(descStr OUT VARCHAR2) IS
  BEGIN
    descStr := 'Checks for a User Account in fnd_user';
  END getTestDesc;

  ------------------------------------------------------------
  -- procedure to report test name back to framework
  ------------------------------------------------------------
PROCEDURE getTestName(name OUT VARCHAR2) IS
  BEGIN
    name := 'fnd_user User account test';
  END getTestName;

  ------------------------------------------------------------
  -- procedure to provide the default parameters for the test case
.
  -- please note the paramters have to be registered through the U
I
  -- before basic tests can be run.
  --
  ------------------------------------------------------------
  PROCEDURE getDefaultTestParams(defaultInputValues OUT JTF_DIAG_I
NPUTTBL) IS
```

```
    tempInput JTF_DIAG_INPUTTBL;
  BEGIN
    tempInput := JTF_DIAGNOSTIC_ADAPTUTIL.initinputtable;
    tempInput := JTF_DIAGNOSTIC_ADAPTUTIL.addInput(tempInput,'USER
ID','SYSADMIN');
    -- tempInput := JTF_DIAGNOSTIC_
defaultInputValues := tempInput;
  EXCEPTION
   when others then
   defaultInputValues := JTF_DIAGNOSTIC_ADAPTUTIL.initinputtable;
  END getDefaultTestParams;

END;
/
```

Declarative Diagnostics

Declarative Diagnostics is a mechanism that allows metadata-based testing of an application product. Here, a product engineer registers the following metadata:

- Metadata for test execution.
- Metadata for validation rules that are used at runtime to determine the success or failure of the test.
- Metadata for reporting that the engine uses at runtime to generate reports. These reports generated by the framework are in a standardized fashion that includes a test run summary and test details.

Seeding of the above metadata is enabled through UI screens that have been provided by Oracle Diagnostics.

Declarative diagnostic tests are not a replacement for Java or PL/SQL tests, but a means of quickly registering setup type tests for your product without writing code. These are supplemental to the Java or PL/SQL diagnostic tests that you write for granular functionality testing of your product.

To use declarative diagnostics, an understanding of the practical details of one's application and familiarity with Oracle Diagnostics is sufficient.

Structure of a Declarative Diagnostic Test

Every Declarative Diagnostic Test is a "Container" for one or more sub-tests that can be of different types. The diagnostics engine will execute each sub-test of the declarative test in the sequence that it was seeded. The success or failure of the overall Declarative Test is determined by the success or failure of each of the sub-tests.

Oracle Diagnostics has also provided UI screens for administering declarative diagnostic tests. For instance, you could re-order the execution sequence of sub-tests, delete obsolete sub-tests, update existing sub-tests, or add sub-tests to an existing declarative test.

Sub-test Types, Metadata Needed, and Use Case Examples

The framework provides the ability to seed five different types of sub-tests. It is important for you to know the nature of each sub-test and how you could use them.

Required Metadata

Each sub-test type has its own metadata needs. However, irrespective of sub-test type, there are some core metadata elements that each sub-test needs. These are listed below:

- **Name:** A short (< 50 characters) user-defined name that distinguishes the sub-test from other sub-tests within the same declarative test.
- **Description:** An explanation of what the objective of this sub-test is and what it does. Be mindful that the description should be meaningful not just for you but also for the end users.
- **Error Type:** The nature of the error if the sub-test fails at the time it is run. Each sub-test can have one of three error types:
  - **Fatal Error:** Implying that in the event of failure of this sub-test, subsequent sub-tests should not run. This also implies that failing of this sub-test was a high severity failure.
  - **Normal Error:** Implying that in the event of failure of this sub-test, subsequent sub-tests should continue to be executed.
  - **Warning Only:** Implying that failing of this sub-test should be flagged for the system administrator's attention. However, severity of this issue is too low to be termed as an error.
- **Error Message:** In the event of an unsuccessful sub-test execution, the report generated should have this error message for the system administrator at the customer's end. Again, please be mindful that this should be meaningful for the end user to be able to understand what the error means.
- **Fix Information:** In the event of an unsuccessful sub-test execution, the report generated should have information on how to resolve the issue for the system administrator at the customer's end. This should be as self-explanatory as possible so that customers can fix issues at their own end without having to contact customer support.

Finally, the sub-test types that can be registered are as follows:

Count

This purpose of this sub-test type is to count records of the table, view, or SQL statement and logically compare that value against a pre-seeded value. For execution metadata, apart from the core metadata mentioned above, this sub-test will need:

- **From Clause:** The name of a table or view.
- **SQL Query or Where Clause:** Either a complete SQL statement of the format "select count(*) from…" if no value is provided for the above (the "From Clause" field). In case a From Clause has been provided, this field can be used to seed a where clause in the format "condition = value".
- **Logical Operator:** See below.
- **Validation Value 1:** A value against which a logical comparison can be made to determine if running the test was a success or failure. For example, if a value of 2 was provided and the logical operator ">" was seeded, success would imply that the generated SQL returned greater than two rows.

- **Validation Value 2:** If the seeded logical operator is "BETWEEN", then two values are needed to make a comparison.

Based on the metadata, the framework will generate and execute a SQL statement. An example of a SQL statement that could be generated is:

```
select count(*) from FND_GRANTS where grantee_key like 'FND_RESP6
90:21841'
```

In the above example, the sub-test is testing to find out the number of records in the FND_GRANTS table that have a grantee_key of FND_RESP690:21841.

Using the Validation Value 1 (say "2") and the logical operator (say ">"), the framework will determine if the number of records generated was greater than 2 or not.

Record

The purpose of this sub-test type is to check if the generated SQL based on the seeded metadata returns any records or not. Developers can seed whether the SQL should generate records or not and based on that, the Diagnostics engine will make the appropriate comparisons.

For execution metadata, apart from the core metadata mentioned above, this sub-test will need:

- **SQL Query:** A complete SQL statement.
- A **Yes** or **No** selection, indicating whether the query should generate rows or not.

An example of a SQL statement that could be generated is for this sub-test type would be:

```
select * from FND_APPLICATION where application_short_name = 'AOL
'
```

In this case, if you seed that no records be returned, the framework raises an error condition in the case that this query returned any records.

Column

This is a more powerful sub-test type, as it allows the handling of complex SQL queries with multiple selects across multiple tables or views. The returned values for the selected columns are compared against the corresponding seeded validation values and logical operators. The sub-test is considered as failed if any one of the returned values does not meet the comparison test.

For execution metadata, apart from the core metadata mentioned above, this sub-test will need:

- **From Clause:** One or more table or view names. For example, **fnd_application a, fnd_responsibility b** if there are multiple tables or views to select from.
- **Select Column Details:** The select clause of the query to be generated uses this metadata. Each column to be selected has the following diagnostic metadata associated:
  - **Column Name:** Name of columns to be selected. For example, **a.application_short_name** or **b.responsibility_key** if there are more than one tables or views in the From Clause.
  - **Logical Operator:** Selected from a list of logical operators.

- **Validation Value 1:** The value with which a comparison for the column's value should be made.
- **Validation Value 2:** The second value if the logical operator chosen is BETWEEN.

- **Where Clause**

An example to illustrate the usage of this sub-test type in the case where the metadata stored by the developer is:

- From Clause: FND_APPLICATION_TL

  Select Column Details:

  Column Name: APPLICATION_ID

  Logical Operator: =

  Validation Value 1: 690

  Where Clause: APPLICATION_SHORT_NAME = 'JTF'

In this case, the framework will generate the following SQL:

```
select application_id from fnd_application_tl where application_s
hort_name = 'JTF'
```

Each record fetched by the query will be compared for equality (=, the logical operator) with 690.

The report generated for this sub-test type will contain the query generated by the engine and the entire result set that the query returned to the engine.

System Parameter

The purpose of this sub-test type is to check if the JVM and system parameters at the customer end have the desired values. In many cases, several product modules require manual setup sub-tests that may require -D parameters. Frequently, customers to not have those values in place. This sub-test makes it convenient to identify such issues.

For execution metadata, apart from the core metadata mentioned above, this sub-test will need:

- System parameter name.
- The desired value.

The diagnostic engine will only make an equality comparison for the retrieved value and the desired value. If they do not match, this sub-test is considered to have failed.

Test Container

The purpose of this sub-test is to eliminate the need to write a diagnostic Java test that is just a container of other diagnostic test cases (called dependencies). The value-add, apart from eliminating the need to write Java code for containers, is that you can add Java and PL/SQL diagnostic test cases that are run sequentially by the diagnostic engine.

The report that is generated for this sub-test is a combination of all the reports generated by the test cases included in this Test Container.

Tests are run in the sequence that they were registered in this sub-test. This is identical to run tests that are dependencies. However, as PL/SQL and Java tests can be run at the

same time, the pipelining of inputs will work only for Java test cases because PL/SQL test cases cannot return output values at this time.

Logical Operators for Comparison

Logical Operators mentioned for all SUB-TEST TYPE definitions above can be one of the following: <, >, <>, <=, >=, =, BETWEEN.

If the chosen logical operator is <, >, <>, <=, >=, or BETWEEN, then the assumption is that the Validation Values are a numeric values.

If the logical operator chosen is BETWEEN, then the values will be compared as numeric values and in the format v1 <= x <= v2, where x is the actual value and v1 and v2 are the seeded validation values 1 and 2.

Integrating LOVs With Diagnostics

This section discusses how to add LOV inputs to diagnostic test cases. For existing test cases, minimal changes are necessary. However, you do need to provide an implementation for a class that extends **QALovAbstract** to add this functionality.

Use the following procedure to determine whether or not to use an LOV input:

1. Determine which inputs are appropriate for LOV use. Typical LOV candidates are inputs that require non-mnemonic values and those that are constrained by other fields.
2. Determine how and where the values of the LOV are to be retrieved. This is usually satisfied by a database query that retrieves particular columns from a database table.
3. Extend **QALovAbstract**.
4. Add the LOV implementation to the test case input.

Implementing an LOV

First, you must have a class that extends the QALovAbstract class.

```
package oracle.apps.jtf.regress.qatool;

import java.util.Hashtable;

public abstract class QALovAbstract implements QALovInterface {

    public abstract String[] getHeaders();

    public abstract String[][] getValues(Hashtable context, String
 filter) throws Exception;

    public abstract int getValueColumnIndex();

    public String getFilterName() {
        return null;
    }

    public String getDescription() {
        return null;
    }

}
```

- **String [] getHeaders()**

  This method determines the names of all the columns that need to be displayed by the LOV. If your test case needs an LOV with two columns, then you should return a String array with two values. However, it could be the case that you do not want to display certain columns to the user. For columns that you want to hide, you can have their names be an empty String. Any columns that have an empty String for a name will not be rendered.

- **String [] getValues(...)**

  This method defines the SQL executions for your LOV data. The framework handles the rendering. Your only tasks are to execute a SQL query/data retrieval step, handle any unexpected errors, and return the ResultSet in the form of a two-dimensional array. The context object will be available for the LOV implementation. It contains all the inputs needed for the test case. If any columns are hidden, their values should still be returned by this API.

- **init getValueColumnIndex()**

  This method defines one of the getHeaders() columns as the index. If your test case input is the Responsibility ID and getHeaders returns two columns (Responsibility Name, Responsibility ID), then this method returns the index of Responsibility ID as it serves as an input to the test case. It is valid to return the index of a column that is hidden. In this case, the first non-hidden column in the LOV is selectable. However, when the user clicks this value, then it will be the value of the hidden column that is passed back to the test case.

- **String getFilterName()**

  Implementing this API is optional. By default, the prompt that is displayed next to the search field is the name of the test input field that this LOV is tied to. However, by implementing this method, that prompt is whatever String this API returns.

- **String getDescription()**

Implementing this API is optional. By default, there is no description displayed to the user. By implementing this method, a developer can provide a description about the LOV for users, such as an explanation of the searchable field or more details about the type of values being displayed.

LOV Provider Sample Code

The following is an example of an LOV implementation. This class can be used for LOV Support for any test case which depends on Responsibility ID as one of its inputs.

```
package oracle.apps.jtf.regress.qatool;
import java.sql.SQLException;
import java.util.Hashtable;
import java.util.Enumeration;
import oracle.apps.jtf.base.resources.FrameworkException;
import oracle.apps.jtf.aom.transaction.TransactionScope;
import oracle.apps.jtf.base.session.ServletSessionManager;
import oracle.apps.jtf.base.session.ServletSessionManager;
import com.sun.java.util.collections.ArrayList;



/**
 * An example of an LOV implementation. It fetches the Responsibil
ity Names and Responsibilty ID's
 * based on the Application and User. It also allows filtering by
Responsibilty Name.
*
*/
public class RespLovImpl extends QALovAbstract {

    public static final String RCS_ID = "$Header: RespLovImpl.java
 115.1 2002/02/18 21:45:28 bsanghav noship $" ;
    String[] heading = {"", "Responsibility Key", "Responsibility
Name"};
    int valueIndex  = 0;
    public final String APPLICATION_ID = "APPLICATIONID";
    public final String USER_NAME = "USERNAME";

     /** Gets the Column headings for the LOV. The call to this me
thod should be made, preferably,
     * after the getValues() method is called.  Note that the firs
t element of the array is the empty
     * string.  This means that the column is hidden from the user
.  However, this column will hold
     * our Resp ID values which will populate the input.
     */
public String[] getHeaders(){
        return heading;
    }

    /** Gives the index of the value column.  Here we are returnin
g "0" which is the index of the
     * hidden column that has the Resp. ID values.
     *
     */
    public int getValueColumnIndex(){
        return valueIndex;
```

```
    }

    public String getDescription() {
        return "Select value by clicking on the Responsibility Key
.  You can restrict the list by giving a partial" +
       " Responsibility Name in the search field along with '%' as
 a wildcard";

    }

    public String getFilterName() {
        return "Responsibility Name";
    }

    public String[][] getValues(Hashtable context, String filter)
throws Exception {
        Hashtable newContext = new Hashtable(context.size());
        Enumeration en = context.keys();
        if (en == null ) throw new FrameworkException("Not enough
data supplied to get LOV.");;
        while(en.hasMoreElements()){
          String key = (String)en.nextElement();
          if (key==null) { throw new FrameworkException("Enumerati
on gave back anull object. Something really wrong"); }
          String newKey = key.toUpperCase();
          newContext.put(newKey,context.get(key));
        }
        String appId = (String)newContext.get(APPLICATION_ID);
        String userName = (String)newContext.get(USER_NAME);
        int iAppId = -999999; // some non existent appid
        if (appId == null) {
          throw new FrameworkException("Application ID not specife
d");
        } else {
          try {
            iAppId = Integer.parseInt(appId);
          } catch (NumberFormatException pe){
            throw new FrameworkException("Could not convert Applic
ation Id into a numeric value");
          }
        }
        if (userName == null){
          throw new FrameworkException("User Name not specifed");
        }
        try {
    return getData(iAppId,userName.toUpperCase(),filter);
        } catch (SQLException sqe) {
          throw sqe;
        } catch (FrameworkException fwe){
          throw fwe;
        }
    }

    private String[][] getData(int appId,
                               String userName,
                               String filter) throws FrameworkExce
ption,SQLException{
                      return getData(appId,userName,filter);
```

```
    }

    private String[][] getData(int appId,
                               String userName,
                               String filter) throws FrameworkExce
ption,SQLException{

        String queryString = "select a.RESPONSIBILITY_ID, a.RESPON
SIBILITY_KEY, a.RESPONSIBILITY_NAME from FND_RESPONSIBILITY_VL a ,
 FND_USER_RESP_GROUPS b , FND_USER c where c.USER_NAME = '" + user
Name + "' and a.APPLICATION_ID = " + appId + " and b.USER_ID = c.U
SER_ID and a.RESPONSIBILITY_ID = b.RESPONSIBILITY_ID and a.RESPONS
IBILITY_NAME like '" + filter +"'";

        String[][] rsString = null;

        rsString = Util.getTable(queryString);
        heading =  Util.getHeaders(queryString);
        return rsString;
    }
}
```

Incorporating LOVs in Diagnostic Test Cases

To incorporate an LOV in a diagnostic test case, you need to make a simple one line change for every input that is to be an LOV input.

The MenuTest has been modified to plug in the LOV support to one of its inputs.

- Before using the LOV feature:

```
private void init() {

  addInput(new QATestInput("Username", "SYSADMIN"));
  addInput(new QATestInput("ApplicationID", new
           Integer(TABAPPID)));
  addInput(new QATestInput("ResponsibilityID",new
          Integer(RESPID)));
  .
  .

}
```

- After using the LOV feature:

```
private void init() {

  addInput(new QATestInput("Username", "SYSADMIN"));
  addInput(new QATestInput("ApplicationID", new
           Integer(TABAPPID)));
  addInput(new QATestInput("ResponsibilityID", new
           RespLovImpl()));
 .
 .

}
```

You can always provide a default value for an LOV input using one of the overloaded methods provided in **QATestImpl**. In this case:

```
addInput(new QATestInput("Responsibility ID", new
         RespLovImpl(), "21841"));
```

Default LOVs

Below are some sample LOV implementations that are provided by Oracle Diagnostics for use in diagnostic test cases.

- **AppLovImpl.java**

  This LOV can be tied to an input that requires an application ID. Since application IDs are not easy to remember, the LOV pairs the application's full name with the application ID in the LOV pop-up window. Users can filter the application ID column using the wildcard character (%).

- **RespLovImpl.java**

  This LOV can be tied to an input that requires a responsibility ID. The LOV requires two parameters to be present in the LOV context hashtable: "Username" and "ApplicationID". In order for this LOV to work, the test must have input parameters with the exact same input names (ignoring case). When given the user name and application ID values, it queries the database for available responsibility IDs with their responsibility names. Users can filter the responsibility ID column using the wildcard character (%).

- **LangLovImpl.java**

  This LOV can be tied to an input that requires a language code from the FND_LANGUAGES table. This LOV has three columns from the FND_LANGUAGES table: LANGUAGE_CODE, LANGUAGE_ID, and NLS_LANGUAGE . Users can filter based on the NLS_LANGUAGE column using the wildcard character (%). Note that this filter is different than the value that will actually be populated by the LOV (i.e. LANGUAGE_CODE).

- **UserLovImpl.java**

  This LOV can be tied to an input that requires a valid user name from the FND_USER table. This LOV pairs the USER_ID and USER_NAME columns from the FND_USER table. Users can filter the values in the USER_NAME column using the wildcard character (%).

PL/SQL LOVs

The following is sample code for a PL/SQL LOV:

```
PROCEDURE getDefaultTestParams(defaultInputValues OUT NOCOPY JTF_
DIAG_INPUTTBL) IS
    tempInput JTF_DIAG_INPUTTBL;
  BEGIN
    tempInput := JTF_DIAGNOSTIC_ADAPTUTIL.initinputtable;
    tempInput := JTF_DIAGNOSTIC_ADAPTUTIL.addInput(tempInput, 'App
lication', 'LOV-oracle.apps.jtf.regress.qatool.AppLovImpl');
    defaultInputValues := tempInput;
  EXCEPTION
   when others then
   defaultInputValues := JTF_DIAGNOSTIC_ADAPTUTIL.initinputtable;
  END getDefaultTestParams;
```

In order to let the Oracle Diagnostics engine catch the LOV input type, the default input value should be in the format starting with: " LOV-", followed by the whole namespace of Java class.

The Diagnostics engine parses the string value starting with "LOV-", and gets the namespace of the class to instantiate the LOV class.

You then implement the Diagnostics LOV java file, following the usage described in Implementing an LOV, page 2-39. Also, you can use the default LOV classes that the Diagnostics framework already has (oracle.apps.jtf.regress.qatool):

- AppLovImpl.class
- RespLovImpl.class
- LangLovImpl.class
- UserLovImpl.class

Oracle Applications Framework Support

Oracle Diagnostics provides APIs to load Oracle Applications Framework Application Modules (the base class is oracle.apps.fnd.framework.OAApplicationModule) within Java test cases based on the Application Module definition name supplied. Once an Application Module has been loaded using Diagnostic APIs, you can introspect ViewObjects created on a standalone basis or pulled out of an Application Module.

The two APIs that have been provided belong to class oracle.apps.jtf.regress.qatool. OABridge and are as follows:

Secure API

```
static public OAApplicationModule getApplicationModule(
    String username,
    String appShortName,
    int respID,
    String amDefName) throws FrameworkException;
```

- This is a secure API that can only be called by a Diagnostic Super User since this instantiates a new user context based on the username supplied.
- It uses the AppsContext of the username supplied to instantiate an ApplicationModule based on the AM Definition Name supplied.
- It can only be called within the runTest method of a Java test case.

Non-secure API

```
static public OAApplicationModule getApplicationModule(
    String amDefName) throws FrameworkException;
```

- This is a regular API that can be called by any user.
- It instantiates the user context of the user that started diagnostics and uses the AppsContext of that username to instantiate an ApplicationModule based on the AM Def Name supplied.
- This too can only be called within the runTest method of a Java test case.

Sample Code

```
public boolean runTest() {

    // app module definition name can be a user input

    OAApplicationModule am
        = OABridge.getApplicationModule(amDefName);

    // OR use the secure API which
    // instantiates a user context

    OAApplicationModule am
        = OABridge.getApplicationModule(username,
            appShortName,
            respID,
            amDefName);

    // Obtain view object from the app module.
    // There are many ways
    // one of them is using a simple query
    // string which can be a query string

    String qstring = "select * from fnd_application";

    // Use the qstring for instantiating the
    // view object

    ViewObject vo =
        am.createViewObjectFromQueryStmt("MyVO", qstring);
    // At this point you can introspect your View objects for vali
dity.

}
```

Instantiation of Diagnostic User Context Within Diagnostic Test Cases

The runTest method of a Java diagnostic test case is run within a new thread that has been spawned off for that purpose. That thread first instantiates a guest user session. However, in many instances, the test case requires the same session as that of the user that launched diagnostics. For achieving that, we are providing further API support within class `oracle.apps.jtf.regress.qatool.QAManager`. This API

can only be called inside the runTest method of the Java diagnostic test case; else an exception will be thrown.

```
public static void initDiagUserContext() throws Exception;
```

Once this API is called, the session within the thread will be identical to that of the session of the user that launched diagnostics.

3

Diagnostic Security

Overview

The need for securing diagnostic test cases stems from the fact that test cases have the ability to diagnose sensitive aspects of an application. As an example, using diagnostic test cases customers can test shopping cart or leasing applications, check the sanity of accounting modules, query database tables for data integrity and then render sensitive information on all those pieces within diagnostic reports. All these operations are highly sensitive and should be carried out by users that have been explicitly authorized.

Key Concepts

Test Group Sensitivity

Every test group that you create will have a sensitivity level associated with it. The supported sensitivity values are low, medium, and high. Sensitivity is a function of the type of tests that a group contains. Tests that can display reports which contain privileged information or perform testing of sensitive aspects a product should NOT be placed in low sensitivity test groups. It is the developer's prerogative to pick a sensitivity level for test groups.

Groups that are marked as medium or high sensitivity can only be executed by users who have the appropriately privileged diagnostic roles assigned to them.

Diagnostic Roles

A diagnostic role determines the activities or tasks that a user can perform on Oracle Diagnostics, such as:

- Running test cases (with different input values if an advanced test).
- Viewing detailed test reports after tests have been run.
- Configuring input values for test cases.
- Adding or deleting test cases and test groups across applications registered with Oracle Diagnostics.
- Viewing historical reports for test runs.

The following are descriptions of the available diagnostics roles:

Diagnostics Super User

Has unrestricted and global privileges for all operations on diagnostics. A user with this role can execute, configure, view reports and setup security for all groups and all applications.

This role is granted to the CRM Foundation application responsibility **CRM HTML Administration** which has been assigned to the user **sysadmin.** Any one who wants to view detailed security screens must log in as a user with that responsibility.

Application Super User

Has unrestricted and global privileges for the application associated with his or her responsibility – that is, they can execute, configure, view reports, and set up security for test groups across his or her own application. However, this role also permits the user to execute and configure inputs for test groups of low and medium sensitivities across other applications.

End User

The user with this role can execute and configure inputs for test groups of low sensitivity only. This user cannot view detailed test reports.

Anonymous User

This is not an explicit role; if none of the user's responsibilities have an association with any of the above three roles, then the user is considered to be an Anonymous user. For such users, the diagnostics engine restricts access to pseudo application "HTML Platform". All other test groups across all other applications are restricted from this user.

Note that access to a test group can be given to any responsibility by means of an explicit grant (using the diagnostic security screens). In the case of an explicit grant to a test group, the user can then execute and configure inputs for tests in that particular test group for which his responsibility has been given the grant. They can also view detailed reports.

The table below lists the different abilities of the different diagnostic roles:

Tasks Available to Diagnostic Roles

Task	Diagnostic Super User	Application Super User	End User	Anonymous User
Use LogViewer tab	Yes	Yes	No	No
Perform security configuration	Yes	For test groups of own application	No	No
Configure applications	Yes	For own application	No	No
View detailed reports	Yes	Yes	No	No
Configure test inputs	Yes	For own application and low-medium sensitivity test groups of other applications	Low sensitivity test groups	HTML Platform only
Send e-mail and print detailed reports	Yes	Yes	Summarized reports	HTML Platform only
HTML Platform	Yes	Yes	Yes	Yes

Underlying Security Infrastructure

Oracle Diagnostics uses Oracle Applications Object Library Data Security as its underlying security mechanism. Oracle Applications Object Library Data Security uses the notion of application responsibilities for administering security. For more information about Oracle Applications Object Library Data Security, refer to the *Oracle Applications System Administrator's Guide*.

A grant is defined as a permission to access a secured entity. Grants cannot be given to users directly. Instead, they are given to responsibilities that are assigned to users. Thus if a grant for executing all tests in the test group "Session Tests" is given to the responsibility "Marketing Online Executive" of the application "Oracle Marketing Online", then all users having this responsibility in their responsibility list will automatically have access to the test group in question. Similarly, if a responsibility has been granted a certain diagnostic role, all users having that responsibility in their responsibility list will have been granted that diagnostic role and will be able to use Diagnostics pursuant to the definition of that diagnostic role.

Security Administration

Securing Test Groups

Test group security can be administered in the following ways:

- Marking them as medium or high sensitivity test groups when they are created, this restricting them to Diagnostic Super Users or Application Super Users.

- Assigning grants to certain responsibilities to have explicit access to test groups irrespective of the sensitivity level of the test group. For this purpose, navigate to Configuration > Groups > Select the appropriate group > Choose "Advanced Security"

  This will take you to the "Group Advanced Security" page, through which you can assign access grants to various responsibilities for the test group in question.

Assigning Diagnostic Roles to Responsibilities

For this purpose, navigate to Configuration > Security > Select the appropriate diagnostic role.

This will take you to the "Role Responsibility Assignment" page, through which you can assign access grants to various responsibilities for the diagnostic role in question.

Session Creation / Switching User Context in Test Cases

Diagnostic tests should not seek a password from the user in order to switch the user context.

The framework's security model provides a mechanism for diagnostic tests to switch user context in a programmatic and secure way. Only users who have the Diagnostics Super User role can execute tests that require switching of user contexts.

Diagnostic tests do not accept a password as input parameter, but accept application short name and responsibility ID (optional) as input parameters.

Within the runTest(..) method of the test, you need to use the username, application short name and optionally the responsibilityID and invoke one of the following two APIs to accomplish the user context switch:

```
/**
  * This API can only be called by a Diagnostic Super User
  * Since this API does not take the resp id as a parameter, it us
es the value set for
  * profile option JTF_PROFILE_DEFAULT_RESPONSIBILITY for the user
name supplied.
  */

QAManager.switchUserContext(String username, String appShortName)
throws InadequatePrivilegeException, Exception;
OR

/**
  * This API can only be called by a Diagnostic Super User
  * This API takes the responsibility ID as a parameter explicitly
  */

QAManager.switchUserContext(String username,  String appShortName,
  int respID)
throws InadequatePrivilegeException, Exception;
```

If the user does not have a Diagnostic Super User Role assigned, then these APIs throw an Inadequate Privilege Exception which should be caught by your runTest(...) method. Also, you should populate the error with fix info and report parameters to

reflect why the SwitchContext operation failed. This is illustrated in the code sample below.

```
/**
 * SAMPLE CODE FOR WHAT SHOULD BE USED IN THE RUNTEST METHOD FOR
 * SESSION CREATION
 */

try {

// THIS IS THE APPROPRIATE WAY TO SWITCH USER CONTEXT
QAManager.switchUserContext(username,
        AppUtil.getAppShortName(tabAppID),
        respID);

}catch(InadequatePrivilegeException e){

                error = "Inadequate privileges to call this test.
“
+ ”Please login using superuser access level “
+ “(Resp: JTF_ADMIN_USER)";

                fixInfo = "";
                report = e.getMessage();
                return false;

}
catch (Exception e) {

                error = "Session cannot be started.";
                fixInfo = "";
                report = e.getMessage();
                return false;

}
```

4

Diagnostics Result Reporting

Overview

The following information is recorded in the database whenever a test is run through Oracle Diagnostics:

- A result, with the test execution information.
- Statistical information, such as how many times a test has been run, the total number of failures, etcetera.

Storing test results in the database allows for querying and analysis of the test information. For example, you could find out which tests have failed in the last 24 hours. The statistical information captures summary information about the health of the system.

Database Failover

If the database is down (that is, cannot be read from or written to), then report entries are stored locally on the file system. All test results for that diagnostic session go to the file system, even if the database comes up at some point in the session. If the database stays up, then the log entries will go to the database the next time a session is started.

Report files are only created if the database is down. In this case, they are generated in the directory specified by the following -D parameter to the JServ:

```
-Dframework.Logging.system.filename=<some writable directory>
```

This directory must be writable. When the database is down, there will be one file per diagnostic session.

Accessing Result Logs

You can view result entries for a tests which have been executed in the current diagnostic session. There are a three ways to access the test result information in the UI:

- Click the test name when viewing all the basic tests in a group.
- Click **View Report** when viewing an test run in advanced mode.
- Click **Report** after a test has been executed in basic or advanced mode.

Note that this result information is from tests that have been executed in the current diagnostic session. If you leave Oracle Diagnostics and then re-enter it, you can no longer see the result entries from prior sessions.

Test result entries are stored in the JTF_DIAGNOSTIC_LOG table. The statistical information is stored in the JTF_DIAGNOSTIC_STATS table.

Purging Result Logs

It is good practice to periodically purge result entries to prevent the table from growing too large. Each test that is run results in a new row in the test result table. These entries are not automatically cleared, so the table will keep growing without bound as more tests are run over time.

The statistics table will not grow very large since its size is based on the number of tests registered, not the number of times these tests have been executed. However, since the statistics information keeps a running record of different metrics (e.g., number of failures) it is important to refresh this information to keep the statistics relevant to the state of the system.

Test result and statistic entries can be removed from the database by navigating to **Configuration > Applications**. The bottom of the screen shows all the result and statistic entries for the application for all sessions. Deleting the result entries, removes all the log entries which have been created for this application, regardless of session. Note that this includes the current diagnostic session.

This page only shows the result entries in the database, and only deletes the log entries in the database. If any result files have been generated by diagnostic sessions (if the database was down), then they have to be manually deleted by going the directory specified by the **-Dframework.Logging.system.filename** parameter.

Scheduling Routine Purging

Deleting Diagnostic Logs

You can use the concurrent program "Delete Diagnostic Logs" to delete all Oracle Diagnostics test reports that are older than a given number of days for one or more applications. The application list is specified as a space-separated list of application short names to the concurrent program. The number of days threshold is specified as another input argument to the concurrent program. A value of "0" causes all logs to be deleted, regardless of age. The program can be executed through Oracle Applications Manager or through the Oracle Forms UI for submitting Concurrent Program Requests.

Deleting Diagnostic Statistics

You can use the concurrent program "Delete Diagnostic Stats" to delete all the statistics that have been collected about Oracle Diagnostics test executions. This program can be run in the same way as "Delete Diagnostic Logs".

Historical Logs: LogViewer

Oracle Diagnostics provides the ability to query the repository of diagnostic test logs in the database. However, access to the LogViewer is restricted to users with the Diagnostic Role "Diagnostic Super User". For details, see Diagnostic Security, page 3-1. To authorized users, the **LogViewer** subtab displays on the **Home** tab. The LogViewer allows users to query for logs based on multiple criteria, including dates, applications, test groups, sessions, test result status, etc. Apart from this, the diagnostic homepage renders a bin which displays test failures from the past week. Authorized users can drill down to detailed failure reports from this bin.

The LogViewer allows users to do the following:

- view one log per page, and drill down to a specific test log
- save a single log or multiple logs (combined in a ZIP file) to local machine and then upload the saved logs to Oracle*Metalink* using the **Save** and **Upload to Support** buttons

Microsoft Excel Reporting for Diagnostics PL/SQL Test Results

If the results of a PL/SQL test represent the results of a database query, then those query results can be converted to Microsoft Excel spreadsheets. A button at the top of the results page for such a test is provided for the conversion. By selecting this button, a user will be directed to Diagnostics Excel Reporting page where the query results are displayed in an Excel format. The Excel spreadsheets can then be uploaded to Oracle*Metalink*.

5

Launching Oracle Diagnostics

Overview

Oracle Diagnostics is available to users in more than one way. This chapter discusses how to access the different user interfaces that are available, as well as what can be accomplished in each of them. In summary, Oracle Diagnostics can be accessed in the following ways:

- Standalone HTML User
- CRM System Administrator Console
- Oracle Applications Manager
- Command-line Console

Standalone Diagnostics

Most Diagnostics users will enter the application using this method.

Access

Function Security and Standalone Diagnostics

Oracle Diagnostics also releases its standalone version, which has the minimum code dependency and can be installed in Oracle Applications Systems 11i.6 to 11i10. This standalone version utilizes Oracle Application Object Library function security.

By default, two responsibilities, CRM HTML Administration and System Administration, have this seeded Diagnostics menu associated with them. A user with the CRM HTML Administration responsibility can access Oracle Diagnostics by selecting the "Diagnostics" link under the "Setup" menu. A user with the System Administration responsibility can access Oracle Diagnostics by selecting the "Diagnostic Tests" link under the "Diagnostics" menu.

In Oracle Diagnostics version 2.4, the Diagnostics JSP pages menu is associated with a new seeded responsibility, Oracle Diagnostics Tool. For details of how to assign this responsibility to user, please see: Diagnostics Responsibility Configuration, Oracle*MetaLink* Note 358831.1. After this assignment, the user should be able to log in via the Oracle Applications Login page and access the Diagnostics JSP pages from this responsibility.

Logging In

The Standalone Diagnostics entry point to Diagnostics can be reached by going to the following URL:

```
http://<host>:<port>/OA_HTML/DiagLogin.jsp
```

This URL will redirect to Applications Login page. After the user logs in, the user will be directed to the Oracle E-Business Suite Navigator page. Oracle Diagnostics can be accessed from the appropriate responsibility.

Features

With the Diagnostics Security Framework in version 2.1 and later, anyone entering Diagnostics using this method is given whatever Diagnostic Role is granted to the Guest User's responsibilities. You should grant the Guest User the "End User" Diagnostic Role if you want the Guest User to be able to do more than work only with the HTML Platform. For details about Diagnostic Roles, see Diagnostic Security, page 3-1.

Typically, this user will not be able to configure Oracle Diagnostics or view detailed test reports. In order to perform these functions, the user needs to access Diagnostics using the CRM System Administrator Console or through Oracle Application Manager.

Diagnostics 2.4 includes the following features:

- Search Tests: Users can search for tests using specific keywords on the Diagnostics home page.
- Batch Processing: A user can choose diagnostic tests across multiple applications and configure input for the tests together as needed, and then run those tests together.
- Diagnostics Test Sets: A test set is a set of tests written within a XML file. Input parameters supplied for the test set is included within the tests themselves. A test set can be directly run without any user interaction of selecting tests and specifying input. A user can:
  - search, view, and run the test sets shipped by Oracle Applications.
  - generate a test set by using the "Batch Processing" pages, and save the test set to a local PC.
  - upload a test set from a local PC to run it directly. The test set itself may be downloaded from Oracle*MetaLink* or generated by the user beforehand.
- After a user executes Diagnostics tests from "Basic Tests" page or "Advanced Tests" page, the user can save the test result and then upload the test report to an Oracle*MetaLink* note. These actions can be performed from the test report page using the **Save** and **Upload to Support** buttons.

  After a user executes Diagnostics tests from "Batch Processing" page, the user can save multiple reports as a ZIP file to a local machine, and then upload the ZIP file to Oracle*Metalink* using the **Save All Reports** and **Upload to Support** buttons, respectively.
- From the LogViewer page, a user can save a single log or multiple logs (together as a ZIP file) to a local machine and upload the file to Oracle*Metalink.*

Bookmarking Pages in the Diagnostics UI

Oracle Diagnostics supports bookmarking of diagnostic pages. Bookmarking is the saving of URLs as "Favorites" or "Bookmarks" in the browser. Since bookmarking captures the URL on the browser, users can quickly navigate to the URL.

Remember that Oracle Diagnostics relies on several session and form post parameters which may not show up on the URL. In cases where the Diagnostic engine can find the relevant information through the URL, the appropriate diagnostic page will be displayed. In all other cases, it will default to the Diagnostics homepage. If the user session has expired, diagnostics will start a guest user session -- the guest user may not have the appropriate responsibility to view the information being sought. In such cases, you should authenticate yourself before using diagnostic-related bookmarks.

CRM System Administrator Console

Administrator-level users that need to configure Oracle Diagnostics or view sensitive data in the detailed test reports should enter using this method or through Oracle Applications Manager. The CRM System Administrator Console can be navigated to through the following URL:

```
http://<host>:<port>/OA_HTML/jtflogin.jsp
```

Clicking the **Diagnostics** tab will launch the Oracle Diagnostics user interface.

Features

Depending on the users' responsibilities, they will be assigned a diagnostics role. In order to have unrestricted access to all diagnostics features such as configuration and viewing test details, a user must have the Super User Diagnostics Role. For details, see Launching Oracle Diagnostics, page 5-1.

Oracle Applications Manager

Select features of Oracle Diagnostics appear in Oracle Applications Manager (OAM). Through the OAM UI, the application administrator can view diagnostic test execution statistics as well as detailed log reports. If the application administrator needs to perform other Oracle Diagnostics functions, he or she can launch the standard Oracle Diagnostics UI from OAM.

Oracle Diagnostics has been integrated with OAM version 2.2 and above.

Finding Oracle Diagnostics in OAM

Log in to Oracle Applications Manager. After successfully logging in, you will reach the OAM dashboard.

The entry point to Oracle Diagnostics is at the **Diagnostics** subtab in the dashboard.

Diagnostics Test Summary

Clicking the **Diagnostics** subtab displays summary information about diagnostic tests executed on the environment using Oracle Diagnostics.

The diagnostic test results are categorized by applications and then by groups within applications. By default, **Failures in Past Week** is displayed. This means that only those tests that failed within the last seven days are shown. You can filter the data by choosing to **View** only the failures within the last 24 hours or see all the tests regardless of when they last failed, or if they even have failed at all. Clicking on the **Expand All** link will display the entire hierarchy.

The **Status** column of the table reflects a rolled-up status for all the entities in the hierarchy. For instance, the status icon for a group is the worst status of all the statuses for the tests in that group, and that for an application is the worst status among all the groups within that application. The status of a test is determined by the last time the test was executed. The **Last Failure Time** and **Last Execution Time** columns have values only for individual tests.

Clicking on the **Status** icon will show the details of that test's last execution.

Refreshing the Summary Data

Clicking on the refresh icon next to the **Last Updated** time will retrieve the latest Diagnostic summary data from the database. Summary data will only be refreshed upon logging into OAM or by explicitly clicking the refresh icon. Also, each view is refreshed separately and has its own **Last Updated** time. For example, refreshing data for the **Failures Today** view will not retrieve new data for the **All** view.

Diagnostic Test Details

This page shows the detailed report generated when the test was last executed. If the test has had a failure, then you can also select to view the last failure by selecting **Last Failure** then clicking **Go**. The information displayed is composed of all the familiar elements Diagnostic log report.

Using the Support Cart

If the issue is a failure that cannot be resolved you can add the details to the OAM Support Cart. The Support Cart lets you store important screen shots that you can include when filing a TAR. To do this, click on the **Add to Support Cart** button at the top of the screen. This will be followed by a confirmation screen stating that the page has been successfully added to the Support Cart. Clicking **OK** will return you back to the test details page.

To see all the items you have placed in the Support Cart. Click the **Support Cart** icon at the very top of the screen. You will see all the screen captures for this session.

Those captures that have the name **oam/diagfwk/testDetails** correspond to Diagnostic test details. Clicking on the **View** icon will show the screen capture. Clicking on the **Save Cart** button will allow you to save the entire cart as a zip file to be included with your TAR.

Launching Oracle Diagnostics from OAM

Clicking on the **Launch Diagnostics** button in either test summary or test details page will pop up a new window containing the full Oracle Diagnostics UI. In the Oracle Diagnostics window, you can run tests and perform all other operations that are normally permitted through Oracle Diagnostics.

Command-line Console

The command-line console is typically used during development or during installation when a system administrator needs to run regression tests to verify if different aspects of the system are still functioning after a patch has been installed.

Naturally, you will not be permitted to access any "Configuration" functionality over the Command-Line Console, since you are implicitly authenticated as a guest user. This is necessary to keep the data registered with the framework secure.

Oracle Diagnostics can be launched in command-line mode using this command:

```
java -DJTFDBCFILE=<dbc file> -Dframework.Logging.system.filename=
<framework logfile>
-Dservice.Logging.common.filename=<service logfile> oracle.apps.jt
f.regress.qatool.QAConsole
```

You need to do the following in this command:

- Specify the .dbc file.
- Specify the locations of the framework and service log files.
- Ensure that the classpath being used has the JAVA_TOP, **jdbc12.zip**, and **jsdk.jar**.

Scheduling Batch Diagnostics

When performing maintenance or verification tasks, you can schedule diagnostic tests to run in batch mode. The concurrent program "Run Diagnostic Tests" is provided for this purpose. You can set up this program through Oracle Applications Manager or Oracle Forms. "Run Diagnostic Tests" can scheduled to execute a single test, a group of tests, or all groups of tests that are registered under an application.

When using Oracle Applications Manager to set up scheduling tests, use the navigation path Site Map > Concurrent Requests > Submit New to access the scheduling wizard. In the first page of the wizard, search for "%Diag%", and select "Run Diagnostic Tests". In the next page use the LOV to select the test application, and optionally, enter group name and test name to schedule the tests.

6

Logging Framework Overview

Overview

The Oracle Applications Logging Framework provides the ability to store and retrieve log messages for debugging, error reporting, and alerting purposes.

You can set up, view, and purge log messages through HTML-based user interface pages that are located in Oracle Applications Manager. For more information about these pages, refer to the *Oracle Applications System Administrator's Guide*or the Oracle Applications Manager online help.

Target Audience

The target audience of this and other chapters related to logging are as follows:

System Administrators

As a system administrator, you should monitor alerts and log messages to manage system activities and troubleshoot problems.

Application Developers and Consultants

You can use this manual to learn how to add alerts and log messages to your code. Also, you can review log messages for debugging purposes.

Key Features

- All Oracle Applications log messages are stored in a central repository.
- Messages can be correlated across middle-tier and database servers.
- Autonomous transactions are used to log to the database.
- Context information is captured internally to facilitate the analysis of messages.
- Configurable System Alerts allow for e-mail and pager notification.
- Messages can have attachments up to 4 GB in size.
- Implementations in Java, PL/SQL, and C.
- Multiple ways to control which messages are logged:
    - Set Oracle Applications Object Library (FND) profiles in the database to turn on logging, based on the application user, responsibility, application, or site.

- Set the logging level to control which messages are logged, based on their severity. There are six severities, ranging from STATEMENT (least severe) to UNEXPECTED (most severe).
- Filter log messages by source module. Use of a wildcard character (%) is supported.
- Turn on logging for individual processes.
- Turn on logging for individual threads within a JVM.

Terminology

Log Message

A complete log message has a set of message identifiers and the actual text of the log message. The only identifiers that developers must provide are the message, module, and severity. Everything else is automatically captured by the Logging Framework.

Log messages include the following:

- Time Stamp: The time the message was recorded.
- Log Sequence: A unique sequence number internally generated for the message.
- User ID: A unique identifier for the application user (foreign key to FND_USER).
- Responsibility ID: The user's current responsibility (foreign key to FND_RESPONSIBILITY).
- Application ID: The user's current application (foreign key to FND_APPLICATION).
- Session ID: A unique identifier for the application user session (foreign key to ICX_SESSIONS).
- Transaction ID: A unique identifier to identify the runtime context of the application. Four different transaction types are currently supported:
  - Concurrent Program (CP): the CP ID (foreign key to FND_CONCURRENT_PROGRAMS), Request-Id (foreign key to FND_CONCURRENT_REQUESTS) are automatically captured.
  - Form Function: the Form ID (foreign key to FND_FORMS), Function-Id (foreign key to FND_FORM_FUNCTIONS) are automatically captured.
  - ICX: the ICX Session ID (foreign key to ICX_SESSIONS) and ICX Transaction-Id (foreign key to ICX_TRANSACTIONS) are automatically captured.
  - Service: the Process ID (foreign key to fnd_concurrent_processes), Concurrent-Queue ID (foreign key to fnd_concurrent_queues) are automatically captured.
- Node: The host name of the machine where the message was generated.
- Node IP Address: The IP address of the machine where the message was generated.
- JVM ID: A unique identifier for the Java process where the message was generated.
- AUDSID: A unique identifier for the database connection (userenv('SESSIONID')).
- Process ID: A unique identifier for the database process (v$session.Process).
- Thread ID: A unique identifier for the thread within the Java process where the message was generated.

- Severity: One of six predefined values that indicate the importance of the message. See the full definition below.
- Module: Represents the source of the message. Typically in Java this is the full class name. When a class name starts with "oracle.apps", then the leading "oracle.apps." is dropped in the logged message. For example: "oracle.apps.jtf.util.Encoder" is logged as "jtf.util.Encoder".
- Message Text: The descriptive body of the message. 4000 bytes is the maximum length currently supported. Please accommodate for multibyte characters appropriately. If additional space is required, then log attachments of up to 4 GB can be added.

Module Filter

A module filter is an optional comma-delimited list of strings that you can configure to perform logging for specific modules. You can use a wildcard (%) if desired. For example: "fnd%, jtf%, store%, cart%".

Severity

Each log message belongs to one of the following six severities (listed from least severe to most severe): 1-STATEMENT, 2-PROCEDURE, 3-EVENT, 4-EXCEPTION, 5-ERROR, or 6-UNEXPECTED.

Logging Level

A logging level is a threshold that you can set to control the logging of messages. You can set the logging level to any of the six severities defined above. When you set a logging level, only messages that have a severity greater than or equal to the defined level are logged. For example, if you set the logging level to 5-ERROR, then logging occurs for messages that are 5-ERROR and 6-UNEXPECTED. If you set the logging level to the lowest severity, 1-STATEMENT, then messages of all six severities are logged.

Logging Configuration Parameters

Overview

The following parameters govern logging:

- AFLOG_ENABLED

  Specifies if logging is enabled or not. The default value is NULL (False).

- AFLOG_LEVEL

  Specifies the logging level. The default value is NULL (Log.UNEXPECTED).

- AFLOG_MODULE

  Specifies which modules are logged. The default value is NULL (%).

- AFLOG_FILENAME

  Specifies the file where middle-tier log messages are written.

These parameters can be set as middle-tier properties, Oracle Applications Object Library (FND) profile option values, or a combination of both. Middle-tier properties are set using Java system properties or C environment variables. The middle-tier settings

take precedence over database settings. This allows you to control logging globally from the database, or locally from the middle tier.

If a parameter is not set as either a middle-tier property or an Oracle Applications Object Library (FND) profile option value, then the default value is used. The middle-tier properties only affect the middle-tier logging, and do not affect the database (PL/SQL) layer logging.

Each log message has an associated module and level, which are determined by the author of the message. Whether a log message is actually logged during an enabled instance (AFLOG_ENABLED=TRUE) depends on how the message's level and module compare to the settings of AFLOG_LEVEL and AFLOG_MODULE. The message's level must be greater than or equal to the value of AFLOG_LEVEL, and the module must match the filter AFLOG_MODULE.

Detailed descriptions of the logging parameters follow.

AFLOG_ENABLED

AFLOG_ENABLED determines if logging is enabled. In the database tier, the possible values are "Y" and "N". In the middle tier, the possible values are "TRUE" and "FALSE".

If AFLOG_ENABLED is set to "FALSE" using middle-tier properties, then no logging occurs in the middle tier. If AFLOG_ENABLED is set to "N" using Oracle Applications Object Library (FND) profiles, then no logging occurs in the database tier.

If AFLOG_ENABLED is set to "TRUE", then log messages of the appropriate level and module will be logged either to the database or to a file. Since parameter values set as middle-tier properties take precedence over values set as database profile option values, logging can be globally enabled or disabled for a specific middle-tier process using properties. For example, to completely disable middle-tier logging in a JVM, use "-DAFLOG_ENABLED=FALSE".

For example:

```
/local/java/jdk1.2.2/bin/java -DAFLOG_ENABLED=FALSE org.apache.js
erv.JServ
```

When AFLOG_ENABLED is set in this way, it overrides any value set using database profile option values.

Likewise, logging can be globally enabled. If "-DAFLOG_ENABLED=TRUE" is used, logging will be enabled, even for users whose database profile option value for AFLOG_ENABLED is "N".

The following table shows how middle-tier parameters take precedence over database profile option values:

Middle Tier Values versus Database Tier Values

Database Tier Value	Middle Tier Value	Result
Y	TRUE	Logging occurs in both the middle tier and the database.
Y	FALSE	Logging occurs in the database only.
N	TRUE	Logging occurs in the middle tier only.
N	FALSE	No logging occurs.

AFLOG_LEVEL

AFLOG_LEVEL specifies the logging level. In order to be logged, messages must have a severity greater than or equal to the value of AFLOG_LEVEL.

Any values set using middle-tier properties take precedence over profile option values set in the database. For example, the logging level could be set to "EXCEPTION" in the system properties as:

```
/local/java/jdk1.2.2/bin/java  -DAFLOG_LEVEL=EXCEPTION org.apache
.jserv.JServ
```

The following table lists the supported logging levels for failure reporting:

Logging Levels for Failure Reporting

Logging Level	Value	Meaning	Audience	Examples
Unexpected	6	Indicates an unhandled internal software failure. (Typically requires code or environment fix.)	System administrators at customer sites, and Oracle development and support teams.	"Out of memory." "Required file not found." "Data integrity error." "Configuration error; required property not set, cannot access configuration file." "Failed to place new order due to DB SQLException. " "Failed to obtain connection for processing request."
Error	5	Indicates an external end user error. (Typically requires end user fix.)	System administrators at customer sites, and Oracle development and support teams.	"Invalid username or password." "User entered a duplicate value for field."
Exception	4	Indicates a handled internal software failure. (Typically requires no fix.)	Oracle development and support teams.	"Profile not found. " "Session timed out." "Network routine could not connect; retrying"

The following table lists the supported logging levels for progress reporting:

Logging Levels for Progress Reporting

Logging Level	Value	Meaning	Audience	Examples
Event	3	Used for high-level progress reporting.	Oracle development and support teams.	"User authenticated successfully." "Retrieved user preferences successfully." "Menu rendering completed."
Procedure	2	Used for API-level progress reporting.	Oracle development and support teams.	"Calling PL/SQL procedure XYZ. " "Returning from PL/SQL procedure XYZ."
Statement	1	Used for low-level progress reporting.	Oracle development and support teams.	"Obtained Connection from Pool." "Got request parameter." "Set Cookie with name, value."

AFLOG_MODULE

Module names have the following form:

```
<application short name>.<directory>.<file>.<routine>.<label>
```

For example: "fnd.common.AppsContext.logOut.begin", where <application short name> = "fnd", <directory> = "common", <file> = "AppsContext", <routine> = "logOut", and <label> = "begin".

The Java framework write methods that take a "Class" or "this" Object as a parameter automatically construct the module from the name of the Class. For example, if a log message is being written from an instance of "oracle.apps.fnd.common.AppsContext.class", then the module name will be "fnd.common.AppsContext". Note that the leading "oracle.apps" is always dropped.

The AFLOG_MODULE parameter is a filter against which the module names of log messages are compared. The percent sign (%) is used as a wildcard character. To select all modules, set AFLOG_MODULE to "%". To only log messages from the Class oracle.apps.fnd.common.AppsContext, set AFLOG_MODULE to "fnd.common.AppsContext%".

AFLOG_FILENAME

The default value is NULL (Log to database. If database logging fails, then log to the default file ./aferror.log).

AFLOG_FILENAME specifies the file where middle-tier log messages are written. If a filename is specified as a middle-tier property, then middle-tier log messages are written to that file. Messages at the PL/SQL layer are always logged to the database table.

If a filename is not specified as a middle-tier property, then the following occurs:

- If the database profile option value for the filename is not set in the database or is inaccessible due to an error, then the log messages are written to the default file (aferror.log).
- If the database profile option value for AFLOG_FILENAME is accessible, then the database value is read.
- If the database profile option value is null, then messages are logged to the database.
- If the database profile option is not null, then messages are logged to the specified file.

If the middle-tier process does not have write permission on the specified file, then it logs to the default file. If it cannot write to the default log file, then it writes log messages to STDERR.

If the full path is not specified in the filename, then the path is considered as relative to the current working directory.

If a separate log file for each middle-tier process is needed, then give each process a middle-tier property value for AFLOG_FILENAME.

AFLOG_ECHO

In addition to the four standard log parameters described above, AFLOG_ECHO is available only in the Java tier. It is used to send log messages to STDERR.

If -DAFLOG_ECHO=TRUE and logging is enabled, then all filtered messages are also logged to STDERR in addition to the configured file or database.

7

How to Configure Logging

Using Middle-tier Properties to Configure Logging

All middle-tier property settings take precedence over profile option settings in the database. Logging setup is often done by setting the Apache JServ system properties in the jserv.properties file. This is a quick way to turn on logging for all sites or users, regardless of the current profile option settings. Middle-tier properties only affect middle-tier settings. They do not affect logging at the PL/SQL layer.

Using Java

You can define Java system properties to control logging for each JVM.

The following examples show how to turn on logging for all modules and levels using Java system properties. We assume that the JVM has write permission for the file "/path/to/apps.log". As needed, you can substitute any other file to which the JVM has write permission.

If you plan to log to a file, it is strongly recommended that you override the default file "aferror.log". This default does not specify a full file path and in some cases may not be writable by the middle tier process. Therefore, you should explicitly specify a value for the parameter AFLOG_FILENAME.

Using Command Line JVM System Properties

To enable logging for an application (for example, MyClass) that is run from the command line, add the following parameter values to the command line:

```
/local/java/jdk1.2.2/bin/java
-DAFLOG_ENABLED=TRUE -DAFLOG_LEVEL=STATEMENT \
-DAFLOG_MODULE=% -DAFLOG_FILENAME=/path/to/apps.log MyClass
```

Using Apache JServ JVM System Properties

To enable logging using Apache JServ JVM system properties, add the following to the jserv.properties file:

```
wrapper.bin.parameters=-DAFLOG_ENABLED=TRUE
wrapper.bin.parameters=-DAFLOG_LEVEL=STATEMENT
wrapper.bin.parameters=-DAFLOG_MODULE=%
wrapper.bin.parameters=-DAFLOG_FILENAME=/path/to/apps.log
```

In this case, the log directory used by JServ is a convenient location for the log file.

Using C

You can define environment variables to control logging for each C process.

The following example shows how to turn on logging for all modules and levels using C environment variables. In these examples, we assume that the C process has write permission for the file "/path/to/apps.log". As needed, you can substitute any other file that the JVM can write to.

Note that the default value of the AFLOG_FILENAME parameter does not specify a full file path. Therefore, in some cases this file may not be writable by the middle-tier process. If you plan to log to a file, it is strongly recommended that you explicitly override the default file "aferror.log". To do so, specify a value for the parameter AFLOG_FILENAME.

```
!#/bin/csh
setenv AFLOG_ENABLED Y
setenv AFLOG_LEVEL STATEMENT
setenv AFLOG_MODULE %
setenv AFLOG_FILENAME /path/to/apps.log
./C-Executable
```

Using Database Profile Options to Configure Logging

You can configure logging by setting database profile options. The following table lists profile option names and sample values:

Database Profile Options

Profile Option Name	User Specified Name	Sample Value
AFLOG_ENABLED	FND: Debug Log Enabled	"Y"
AFLOG_MODULE	FND: Debug Log Module	"%"
AFLOG_LEVEL	FND: Debug Log Level	"ERROR"
AFLOG_FILENAME	FND: Debug Log Filename	"/path/to/apps.log"

The available levels are Site, Application, Responsibility, and User. User settings override Responsibility settings, Responsibility settings override Application settings, and Application settings override Site settings.

To emphasize this point, the following is a summary of the impacts of the different profile option levels:

- User: Affects only the given user.
- Application: Affects all users for the specific application.
- Responsibility: Affects all users in any application for that responsibility.
- Site: Affects all users, applications, and responsibilities.

> **Note:** When setting up logging at the Site level, we strongly recommend that you set the logging level to UNEXPECTED. ERROR or EXCEPTION are also possibilities. We strongly discourage

setting the logging level for a site to anything other than UNEXPECTED, ERROR, or EXCEPTION.

Using Logging to Screen

In addition to the above methods where log messages are written to a file or the database, Logging to Screen provides:

- The ability to enable logging on a per HTTP request or per HTTP session basis.
- Dynamic configuration which does not require restarting any servers or changing any log profiles.
- A convenient lightweight mechanism to diagnose performance issues. Each message is timestamped to the millisecond.

If Logging to Screen is enabled, then the Java log messages generated for a particular HTTP Request-Response are buffered in memory and appended to the end of the generated HTML page.

This feature does not affect any existing configurations of file or database logging. File or database logging continues to behave per the configured middle tier log properties and/or log profile values.

Note that this mechanism currently provides only Java layer messages. Regular file or database logging should be used if messages from other layers (e.g., PL/SQL) are needed.

Enabling Logging to Screen in Oracle Application Framework Pages

For security reasons, this feature is only accessible if the "FND: Diagnostics" Profile is set to "Yes".

Use the following procedure to enable Logging to Screen in pages based on the Oracle Application Framework:

1. Click the **Diagnostics** button.
2. Select **Show Log to Screen** from the drop-down list.
3. Choose an appropriate log level.
4. Optionally, enter a module filter criteria such as **jtf***. [In URLs, use the asterisk symbol (*) as a wildcard character, not the percent sign (%).]

Enabling Logging to Screen in CRM Technology Foundation Pages

For security reasons, this feature is only accessible if the "FND: Diagnostics" Profile is set to "Yes".

To enable logging to screen in pages based on the CRM Technology Foundation, append the following to the page's URL:

jtfdebug

Specify the logging level that should be displayed on the current screen.

jtfdebugfilter

(Optional) If desired, this parameter can be used as a filter to display messages based on a Java package name.

For example: `<current_url>&jtfdebug=STATEMENT&jtfdebugfilter=jtf*`

[In URLs, use the asterisk symbol (*) as a wildcard character, not the percent sign (%).]

Startup Behavior

At startup, applications do not have access to profile values. If middle-tier properties are not set, then at startup, the system defaults to logging as follows:

- Logs are stored in the file aferror.log (in the current directory).
- Logs are stored at the level UNEXPECTED.
- Logs are stored for all modules.

After a connection to the database has been established, the site-level log profiles are read. When the user, responsibility, and application have been established, the Oracle Applications Object Library (FND) profiles are read for that user.

For Java and PL/SQL applications, the logging system is initialized by FND_GLOBAL.INITIALIZE (which is called from APPS_INITIALIZE), which is called normally as part of the startup of every Java application session, form, report, or concurrent program. At that point, it has user information and will log with the proper user profiles. Before the FND_GLOBAL.INITIALIZE, if the logging system is called it will self-initialize and log with the site- level profile values.

For Java applications, this is the sequence of startup steps:

1. If any of the log parameters are set as Java system properties, then use them.
2. Logging is not disabled using the Java system property AFLOG_ENABLED=FALSE, and if any of the remaining log parameters are not set as system properties, then retrieve the corresponding Oracle Applications Object Library (FND) profile option values from the database. User-level profile values override responsibility-level profile values, which override application-level profile values, which override site-level profile values.
3. If any of the log parameters are not set either as system properties or as profile values (or they are not accessible due to an error), then use the default values.

8

Logging Guidelines for System Administrators

Overview

Set up your system for logging according to the following guidelines. We recommend that you use Oracle Applications Manager as the user interface for any log management tasks.

> **Note:** To optimize performance, if your Oracle Applications system is at Release 11.5.9 or earlier, then it is recommended that you fully disable logging. Instructions for disabling logging are provided in this chapter.

Recommended Default Site-Level Settings

For normal operations, we recommend that you configure your system as follows:

- Enabled: On
- Logging Level: UNEXPECTED
- Log Repository: Database
- Module Filter: %

> **Caution:** If you set the default site-level logging level to STATEMENT or PROCEDURE, a decrease in system performance could result. Under that configuration, the large amount of generated log messages might significantly slow down the system. Furthermore, if the site-level logging level is set to a low severity for a long time, then the FND_LOG_MESSAGES table could potentially run out of space.

Recommended Settings for Debugging

If you need to lower the logging level in order to gather information about a system error, use the following recommended configurations. (As stated above, the default logging level should be UNEXPECTED. This maintains optimum system performance.)

Using Logging to Screen

For Java-based pages that are based on the Oracle CRM Technology Foundation or the Oracle Applications Framework, if you have access to the browser that is displaying the

generated HTML, you can use the Logging to Screen feature to view further details if an error is reported. See: Using Logging to Screen, page 7-3.

This lightweight mechanism works best in cases where:

- You are interested in Java layer messages only.
- Debugging of is required for a particular request-response. For example, a JSP request from a browser.
- Debugging is required for all request-responses within a specific session.

Pinpointing an Error to a Specific User

You can use Oracle Application Object Library profiles to enable logging for the specific user, responsibility, and application that were active when the error occurred. Ask the user to log in again for the profile changes to take effect. Remember to return the profiles to their usual values after debugging has been completed.

If you suspect that certain code is causing the problem, then use hierarchical module filters to restrict which messages are logged. For example: fnd.common.%

Set the logging level according to the appropriate level of detail. Recall that EVENT messages report key progress events, while EXCEPTION, ERROR, and UNEXPECTED messages report failures.

For High Volumes

For high load, high volume scenarios, you can log middle-tier messages to a local file, which is faster than logging to a remote database. To do so, define the AFLOG_FILENAME property to write all middle tier logging to a local file. Be sure to limit the number of generated messages:

- Use Oracle Applications Object Library FND Profiles to restrict logging according to:
  - Specific users
  - Specific responsibilities
  - Specific applications
- If you suspect that certain code is causing the problem, then use hierarchical module filters to restrict which messages are logged. For example: fnd.common%
- Set the logging level according to the appropriate level of detail. Recall that EVENT messages report key progress events, while EXCEPTION, ERROR, and UNEXPECTED messages report failures.

For maintenance purposes, you should periodically rotate log files and purge old messages from the database table.

Updating Configuration Properties

If you have configured logging using middle-tier properties, then you must restart the affected processes for those processes to use any modified logging properties.

By default, the Oracle Applications Object Library (FND) log database profile values are cached by each process for performance.

In Release 11.5.9 and later, a cache invalidation mechanism is provided. Thus for all user, responsibility, application, or site profiles, asking the user to log in again typically forces the process to re-read the modified profile values.

Note that the **Logging to Screen** feature does not require JVMs to be restarted, as it does not use any middle-tier or database profile values.

How to Completely Disable Logging

Use the following procedure to completely disable logging:

- If logging is configured using middle-tier properties, then set the AFLOG_ENABLED middle-tier properties to FALSE in all appropriate middle-tier configuration files (for example, jserv.properties) and/or startup scripts.
- If logging is configured using Oracle Application Object Library profiles in the database, then use the logging setup screen in Oracle Applications Manager to turn off logging for all applications, responsibilities, and users. For details, see the *Oracle Applications System Administrator's Guide* or the Oracle Applications Manager online help.

See the "Updating Configuration Properties" section above for details on how and when the modified values come into effect.

Purging Log Messages

You should periodically delete old log messages to account for the space limitations of the database table. In addition, you should periodically rotate log files.

There are several ways to purge log messages. They are described below:

Using a Concurrent Program

The concurrent program "Purge Debug Log and System Alerts" (Short name: FNDLGPRG) is the recommended way to purge messages. This program purges all messages up to the specified date, except messages for active transactions (new or open alerts, active ICX sessions, concurrent requests, and so on). In Release 11.5.10, this program is by default scheduled to run daily and purge messages older than 7 days. Internally this concurrent program invokes the FND_LOG_ADMIN APIs, which are described later in this document.

Using Oracle Applications Manager

In Release 11.5.9 and later, go to **System Alerts and Metrics** from the **Navigate to** drop-down list on the Applications Dashboard. Then click **Logs**. Refer to the Oracle Applications Manager online help for instructions on how to use this screen.

Using the Oracle CRM System Administrator Console

In Release 11.5.8 and later, navigate to **Settings > System > Debug Logging**.

Using PL/SQL

You can use the FND_LOG_ADMIN PL/SQL package to delete log messages.

For example:

```
SET SERVEROUTPUT ON
declare
    del_rows NUMBER;
BEGIN
del_rows := fnd_log_admin.delete_all;
DBMS_OUTPUT.PUT_LINE(del_rows || ' rows deleted');
END;
```

Viewing Log Messages

This section summarizes the different user interfaces that can be used to view and work with log messages, and how to access log messages from each UI.

CRM System Administrator Console

In Release 11.5.8 and later, navigate to **Settings > System > Debug Logging**.

Oracle Applications Framework Pages

In Release 11.5.10 and later, when working in Oracle Applications Framework pages, you can use the following procedure to view log messages.

1. Pages based on the Oracle Applications Framework have a global button labeled **Diagnostics**. Click this button to open a window where you can choose **Show Log**. (Note that this "Diagnostics" global button does not refer to the Diagnostics feature in Oracle Applications Manager that enables management and execution of diagnostic tests.)
2. Select **Show Log** to open the Logs page within Oracle Applications Manager. The Logs page is part of the System Alerts and Metrics feature.

> **Note:** For the Diagnostics global button to be visible, the profile option FND_DIAGNOSTICS must be set to YES.

Oracle Applications Manager

In Release 11.5.9 and later, go to **System Alerts and Metrics** from the **Navigate to** drop-down list on the Applications Dashboard. Then click **Logs**.

Oracle Forms

Navigate to **Help > Diagnostics > Logging**.

9

Logging Guidelines for Developers

Overview

You should utilize logging APIs frequently throughout your components. This will aid in localizing problems if a bug is reported. We recommend that you carefully select where you place logging calls, keep your code readable, and put only useful and necessary information into log messages.

The log message text, module source, and severity come directly from you through the coding of the APIs. These three fields cannot be changed or amended other than through the code, so aim to make the information as informative and concise as possible.

As a developer, you only need familiarize yourself with a few APIs and the six severities. Call the appropriate API and pass in the following three fields:

- Module Source
- Severity
- Message Text

All other fields are automatically populated by the APIs.

APIs

The following APIs are used to write log messages to the database:

- The FND_LOG PL/SQL package.
- The oracle.apps.fnd.common.AppsLog Java class.
- The aflog(*) C APIs.

When the APIs write to the database, they typically communicate with the package FND_LOG_REPOSITORY. This package is what actually writes the messages. When debugging middle-tier edge cases, log messages can be sent to a file.

Handling Errors

Use the following procedure to handle errors in your code:

Step 1: Log internal error details (for example: Exception Stack Trace, relevant state information). These details will be used by system administrators, support personnel, etcetera to diagnose the issue.

Step 2: If the error is severe, then raise a System Alert to notify the system administrator.

Step 3: If the error affects the end user, report the error to the end user through the UI (or through a Request Log in the case of a concurrent program). The message should be a translatable user-friendly message, and should not contain any internal error details.

Performance Standards

For performance reasons, you are required to check if logging is enabled for the severity of your message. This should happen before you create any objects or concatenate any strings that form your log message. Checking the value of an integer is less costly than allocating objects or concatenating strings. Remember that function arguments are constructed before the function call. That is, a string concatenation would occur before the Log write*(..) call is made! You should explicitly check if logging is enabled to prevent string creation when logging is disabled.

Sample Java Code

```
if( AppsLog.isEnabled(Log.EVENT) )
  AppsLog.write("fnd.common.WebAppsContext", str1 + str2, Log.EVEN
T);
```

Sample PL/SQL Code

```
if( FND_LOG.LEVEL_PROCEDURE >= FND_LOG.G_CURRENT_RUNTIME_LEVEL )
then
    FND_LOG.STRING(FND_LOG.LEVEL_PROCEDURE,
        'fnd.plsql.MYSTUFF.FUNCTIONA.begin', 'Hello, world!' );
end if;
```

Furthermore, you can use a local variable when inside a tight loop or branch of code that is not sensitive to a context switch for the life of the branch. This avoids accessing a package global variable, which is more expensive than a local variable. See the following example:

```
procedure process_rows ()
  l_debug_level number:=FND_LOG.G_CURRENT_RUNTIME_LEVEL;
  l_proc_level number:=FND_LOG.LEVEL_PROCEDURE;
   begin
          for loop
              validation...
              other calls...
               if ( l_proc_level >= l_debug_level ) then
                   fnd_log....
              end if;
          end loop;
    end;
```

Use a similar optimization for Java and C code wherever possible.

> **Note:** Changes in the Oracle Application Object Library Session (for example, switching responsibilities) can cause the Log Profile values to change. In such scenarios, the Oracle Application Object Library will correctly update FND_LOG.G_CURRENT_RUNTIME_LEVEL, and corresponding values in C and Java as well. However, if you have cached the value in your code, you may not see this change.

Module Source

The Module Source is a hierarchical identifier for a particular code block. The main purpose of the Module Source is to:

- Uniquely identify the source of the message.
- Allow the system administrator to enable logging in particular areas, based on this hierarchy.

For consistency, all module names must be constructed according to the following rules. The module name identifier syntax is as follows:

```
<application short name>.<directory>.<file>.<routine>.<label>
```

Each syntax component is described in detail below.

<application short name>

Indicates the owner application of the code block, specified in lowercase. For example: fnd, jtf, wf, sqlgl, inv.

<directory> | <package>

Indicates the directory or package where the file lives. In general, this is the actual file system directory name. Usually the directory has just one component, but in some cases, it may have two or more components. In Java, this is the full package name. See the following table for examples with their languages and formats.

Examples of File Locations

Language	Format	Example
Java	dir[.subdir]	commonfunctionSecurity.client
C	<src>.dir	Src.flex
Library PL/SQL	Resource	Resource
Forms PL/SQL	Forms	Forms
Reports PL/SQL	Reports	Reports
Server PL/SQL	Plsql	Plsql
Loaders	Loaders	Loaders

<file> | <Class>

Indicates the patchable entity (file) that contains the code. In the case of server PL/SQL, this is the package name rather than the file name. In Java, it is the class name. See the following table for examples with their languages and formats.

Code Entity Examples

Language	Format	Example
Java	<ClassName>	WebAppsContext
C	<filename>	Fndval
Library PL/SQL	<library name>	FNDSQF
Forms PL/SQL	<form filename>	FNDSCAPP
Reports PL/SQL	<report filename>	FNDMNMNU
Server PL/SQL	<packagename>	FND_GLOBAL
Loader	<section>	Afsload

<routine>

Indicates the code routine, procedure, method, or function. In the case of Oracle Forms or Oracle Reports code where there is no routine name, this may be the trigger name. See the following table for examples with their languages and formats..

Routine Examples

Language	Format	Example
Java	<method>	ValidateSession
C	<function>	Fdfgvd
Library PL/SQL	<package.function>	FND_UTILITIES.OPEN_URL
Forms PL/SQL	<package.function>	BLOCK_HANDLER.VAL IDATE_NAME
Forms PL/SQL	<function>	DETERMINE_NEXT_BLOCK
Forms PL/SQL	<trigger>	PRE-FORM
Reports PL/SQL	<function>	LOOKUP_DISPLAY_VALUE
Reports PL/SQL	<trigger>	BEFORE_REPORT
Server PL/SQL	<function>	INITIALIZE
Loader	<action>_<entity>	UPLOAD_FUNCTION

<label>

Is a descriptive name for the part within the routine. The major reason for providing the label is to make a module name uniquely identify exactly one log call. This allows support analysts or programmers to know exactly which piece of code produced your message, without needing to look at the message (which may be translated). Therefore, you should make labels for each log statement unique within a routine.

For grouping a number of log calls from different routines and files that can be enabled or disabled automatically, a two-part label can be used. The first part is the functional group name, and the second part is the unique code location.

For example, Oracle Applications Object Library (FND) descriptive flexfield validation code might have several log calls in different places with labels, such as:

- desc_flex_val.check_value
- desc_flex_val.display_window
- desc_flex_val.parse_code

These could all be enabled by setting the module as "fnd.%.desc_flex_val.%", even though they may be in different locations in the code.

Messages logged at the PROCEDURE level should use the label "begin" for the message logged upon entering and "end" or some variation thereof (like "end_exception") for the message logged upon exiting. For example: begin, end, lookup_app_id, parse_sql_ failed, or myfeature.done_exec.

Module Name Standards

Use the guidelines below to ensure that your code meets the requirement for unique module names across all applications.

- A dot (.) must be used as the separator in the module name hierarchy.
- At minimum, a module name must include the following required components: <application short name>.<directory>.<file>.
- The module name cannot contain spaces or commas. Space and comma characters are reserved for internal parsing. Specifically, nothing except mixed case alphanumeric characters, underscores, dashes, and the dot separator are allowed.
- The module name is compared to without regard to case, so use the same upper, lower, or mixed case format as the directories, files, and routines that the module name is based on. For components that aren't natively upper or lower case (like the application short name and label), use lowercase.

  Be aware that system administrators can turn on debugging at different levels by using the above hierarchy schema. For example, the debug log calls for fnd.plsql.FND_GLOBAL.APPS_INITIALIZE.init_profiles would be enabled if the runtime user enabled logging at any of the following modules:

  - fnd
  - fnd.plsql
  - fnd.plsql.FND
  - fnd.plsql.FND_GLOBAL
  - fnd.plsql.FND_GLOBAL.APPS_INITIALIZE
  - fnd.plsql.FND_GLOBAL.APPS_INITIALIZE.init_profiles

Module Name Examples

- fnd.common.WebAppsContext.validateSession.begin
- fnd.common.WebAppsContext.validateSession.end

- fnd.src.dict.afdict.afdget.lookup_shortname
- fnd.flex.FlexTextField.getSegmentField.lookup_value
- fnd.plsql.FND_GLOBAL.APPS_INITIALIZE.init_profiles
- fnd.resource.FNDSQF.FND_UTILITIES.OPEN_URL.find_browser
- fnd.loaders.afsload.DOWNLOAD_FORM.check_developer_k
- fnd.forms.FNDSCSGN.FND_DATA_TABLE.GET_DB_WINDOW_SIZE.geometry

Severities

For a table that summarizes the available log severities and their usage, refer to the section AFLOG_LEVEL, page 6-5.

STATEMENT and PROCEDURE are intended for debugging by internal Oracle development only. The higher severities, EVENT, EXCEPTION, ERROR and UNEXPECTED, have a broader audience. We encourage you to monitor and attempt to resolve ERROR and UNEXPECTED messages.

Log all internal and external failure messages at EXCEPTION, ERROR, or UNEXPECTED. ERROR and UNEXPECTED messages should be translatable Message Dictionary messages.

Determining where to insert log messages can be an iterative process. As you learn more about your code usage, you gain a better understanding of where to insert log messages that would quickly help isolate the root cause of the error. At a minimum, you should log messages for scenarios described in the next sections.

UNEXPECTED

This severity indicates an unhandled internal software failure which typically requires a code or environment fix.

Log any unrecoverable errors that could occur in as UNEXPECTED. Be very selective in using the UNEXPECTED severity in Message Dictionary-based messages, as messages logged with this severity can be automatically propagated to system administrators as System Alerts. While all log messages should be concise and meaningful, UNEXPECTED messages in particular should be thoughtfully created and reviewed so system administrators can quickly understand the error.

ERROR

This severity indicates an external end user error which typically requires an end user fix.

Log all user error conditions as ERROR. System administrators may choose to enable logging for ERROR messages to see the errors their users are encountering.

ERROR messages should use the Message Dictionary and be seeded in FND_NEW_MESSAGES. If the corresponding error is encountered during runtime, the message must be logged, and if applicable, displayed appropriately. For details, please see the section Automatic Logging and Alerting for Seeded Message Dictionary Messages, page 9-10.

Include the following in ERROR and UNEXPECTED messages:

- Cause: A message describing the cause of the error, and any appropriate state variable values. For example, "Invalid user=" + username;
- "Fix Information" or "Workaround", if known. For example, "Please check your username and/or password."

EXCEPTION

This severity indicates a handled internal software failure which typically requires no fix.

Java exceptions should always be logged. Java exceptions are never part of the normal code flow, and hence should never be ignored. Exceptions should be handled appropriately in your code, and logged for debugging purposes. Whenever you raise an exception, log the cause of the exception first. Convenience log APIs are provided to allow you to pass an exception object in place of the message text. If no severity is passed, then Java exceptions are by default logged at severity EXCEPTION.

Severe exceptions that prevent your product from functioning should be logged at severity UNEXPECTED. For example, log a SQLException when a user places a new order as UNEXPECTED.

EVENT

This severity is used for high-level progress reporting. These apply to application milestones, such as completing a step in a flow, or starting a business transaction.

Whenever your application code reads configurable values, the configured values must be logged. The value may be obtained from profiles, already known attributes of an object (for example, the customer's primary address), defaulting rules, and so on. Log the source, name, and value. For consistency, the label within the module field of such messages should be appended with ".config". For example, "fnd.common.MyClass.MyAPI.config"

PROCEDURE

This severity is used for API-level progress reporting.

Log key functions and APIs as PROCEDURE. The module name for such messages should contain the function or API name, "begin" at the beginning of the procedure, and "end" at the end. For example, the validateSession(..) API is a key API that logs a message at the beginning of the API with module name, "fnd.common.WebAppsContext.validateSession.begin", and the end, "fnd.common.WebAppsContext.validateSession.end".

Whenever you override any base class methods, you must log a message in your derived class's implementation.

The message body should contain the key input values, state values, and return values. For example, log input and output for all controllers, Request, FormRequest, FormData methods.

Log messages at integration points, especially when calling another application's API. Also, use logging when calling procedures across application layers. For example, when calling a PL/SQL API from the Java layer.

STATEMENT

This severity is used for low-level progress reporting.

If you generate SQL (dynamic SQL), it must be logged.

Log all bind variables.

Any user interface choice or dynamic modification of the user interface must be logged. For example, use of "switcher" beans, or page forwards.

Where appropriate, include relevant state variables.

Large Text and Binary Message Attachments

In Release 11.5.9 and later, you can use Message Attachment APIs to add additional context information to log messages and/or System Alerts. This feature provides efficient buffered writing APIs for logging large attachments. The seeded message text for such attachments should contain a brief description of the error, and the attachment should contain all relevant context details.

Currently attachments are stored in a database LOB. As of Release 11.5.10 (specifically, with minipack OAM.H), you can view attachments through Oracle Applications Manager.

Java Code

```
oracle.apps.fnd.common.AppsLog:
    getAttachmentWriter(String, Message, int); // For text data
    getBinaryAttachmentWriter)String, Message, int, ...); // For b
inary data
```

For example:

```
if(alog.isEnabled(Log.UNEXPECTED))
{
  AttachmentWriter attachment = null;
  Message Msg = new Message("FND", "LOGIN_ERROR");
  Msg.setToken("ERRNO", sqle.getErrorCode(), false);
  Msg.setToken("REASON", sqle.getMessage(), false);
  try
  {
    // 'alog' is instance of AppsLog (not anonymous)
    attachment = alog.getAttachmentWriter(
"fnd.security.LoginManager.authenticate", Msg, Log.UNEXPECTED);
    if ( attachment != null )
    {
      // Write out your attachment
      attachment.println("line1");
      attachment.println("line2");
      attachment.println("line3");
    }
  } catch (Exception e)
  {
    // Handle the error
  } finally
  {
    // You must close the attachment!
    if ( attachment != null )
      try { attachment.close(); } catch (Exception e) { }
  }
}
```

PL/SQL Code

```
FND_LOG.MESSAGE_WITH_ATTACHMENT(..);
FND_LOG_ATTACHMENT.WRITE(..); // For text data
FND_LOG_ATTACHMENT.WRITE_RAW(..); // For binary data
```

For example:

```
if( FND_LOG.LEVEL_UNEXPECTED >=
    FND_LOG.G_CURRENT_RUNTIME_LEVEL) then
  FND_MESSAGE.SET_NAME('FND', 'LOGIN_ERROR'); -- Seeded Message
  -- Runtime Information
  FND_MESSAGE.SET_TOKEN('ERRNO', sqlcode);
  FND_MESSAGE.SET_TOKEN('REASON', sqlerrm);
  ATTACHMENT_ID := ND_LOG.MESSAGE_WITH_ATTACHMENT(FND_LOG.LEVEL_UN
EXPECTED, 'fnd.plsql.Login.validate', TRUE);
  if ( ATTACHMENT_ID <> -1 ) then
    FND_LOG_ATTACHMENT.WRITELN(ATTACHMENT_ID, "line1");
    FND_LOG_ATTACHMENT.WRITELN(ATTACHMENT_ID, "line2");
    FND_LOG_ATTACHMENT.WRITELN(ATTACHMENT_ID, "line3");
    -- You must call CLOSE
    FND_LOG_ATTACHMENT.CLOSE(ATTACHMENT_ID);
  end if;
end if;
```

Automatic Logging and Alerting for Seeded Message Dictionary Messages

Seeded Oracle Applications Object Library Message Dictionary messages can be made automatically loggable and automatically alertable by setting the corresponding message metadata attributes.

At runtime, when the Oracle Applications Object Library Message Dictionary APIs are invoked to retrieve these messages in translated format, they will also be internally logged or alerted if the current log configuration permits it.

To be automatically logged, the seeded message's "Log Severity" attribute must be greater than or equal to the configured log level.

To be automatically alerted, the seeded message's "Alert Category" and "Alert Severity" attributes must be defined, and the log configuration should be enabled at least at the 6-UNEXPECTED level.

General Logging Tips

- Do not log sensitive information such as passwords or credit card numbers in unencrypted plain text.
- For readability, do not code the integer values (1, 2, 3, etc.) in your calls to designate severity. Always use the appropriate descriptive name listed above.

How to Log from Java

AppsLog is the class that provides core logging functionality. The Oracle CRM Technology Foundation provides convenient wrapper APIs around AppsLog. This section describes how to use AppsLog and the wrapper APIs.

Core AppsLog

In Java, the core Oracle Applications Object Library (FND) logging functionality is provided by the oracle.apps.fnd.common.AppsLog class. A number of convenience wrappers are available.

AppsLog is a thread-safe class that allows multiple users and threads to log messages concurrently. AppsLog objects are typically created and configured based on a user's log profile settings during the initialization of a user's Oracle Applications Object Library session. Note that AppsLog is not a static singleton class. As different users can have different log profile settings, multiple AppsLog objects will exist within a JVM.

Take care to use the correct AppsLog instance, as there can be multiple concurrent threads and users. Try first to use the current user's AppsContext, and call getLog() on it to get the AppsLog instance. AppsContext's AppsLog is fully initialized based on the current user's log profile settings and Java system properties. Depending on its configuration, it can log to either the database or a file. Do not create static references to this fully initialized AppsLog. Use APIs to get the appropriate AppsContext's AppsLog instance every time.

In edge-case scenarios (for example, before an Oracle Applications Object Library Session is fully initialized and there is no AppsContext available), you can call static AppsLog.getAnonymousLog() to get a standalone AppsLog that is anonymous, initialized only based on Java system properties, and can log only to a file.

Code Sample

```
public boolean authenticate(AppsContext ctx, String user, String
passwd)
    throws SQLException, NoSuchUserException {
    AppsLog alog = (AppsLog) ctx.getLog();
    if(alog.isEnabled(Log.PROCEDURE))  /*To avoid String Concat if
 not enabled */
    alog.write("fnd.security.LoginManager.authenticate.begin",
        "User=" + user, Log.PROCEDURE);
    /* Never log plain-text security sensitive parameters like pas
swd! */
    try {
        validUser = checkinDB(user, passwd);
    } catch(NoSuchUserException nsue) {
        if(alog.isEnabled(Log.EXCEPTION))
        alog.write("fnd.security.LoginManager.authenticate",nsue,
Log.EXCEPTION);
         throw nsue; // Allow the caller Handle it appropriately
        } catch(SQLException sqle) {
        if(alog.isEnabled(Log.UNEXPECTED)) {
        alog.write("fnd.security.LoginManager.authenticate", sqle,
 Log.UNEXPECTED);
       Message Msg = new Message("FND", "LOGIN_ERROR"); /* System
 Alert */
          Msg.setToken("ERRNO", sqle.getErrorCode(), false);
          Msg.setToken("REASON", sqle.getMessage(), false);
          /* Message Dictionary messages should be logged using wr
iteEncoded(..)
            * or write(..Message..), and never using write(..Strin
g..) */
        alog.write("fnd.security.LoginManager.authenticate", Msg,
Log.UNEXPECTED);
    }
    throw sqle; // Allow the UI caller to handle it appropriately
    } // End of catch(SQLException sqle)
    if(alog.isEnabled(Log.PROCEDURE)) /* To avoid String Concat if
 not enabled */
        alog.write("fnd.security.LoginManager.authenticate.end", "
validUser=" + validUser, Log.PROCEDURE);
    return success;
    }
```

OAPageContext and OADBTransaction APIs

The classes oracle.apps.fwk.util.OAPageContext and oracle.apps.fwk.util.OADBTransaction delegate log calls to the AppsLog class. To make logging calls in a UI controller, use OAPageContext. To make logging calls from an application module, use OADBTransaction.

The following are the main logging APIs provided:

isLoggingEnabled(int logLevel)

This returns true if logging is enabled for the given log level. In all cases, test that logging is enabled before creating a message and calling the writeDiagnostics method.

writeDiagnostics(Object module, String messageText, int logLevel)

This writes log messages to the database. Remember that each log message includes a log sequence, user ID, session ID, module identifier, level, and message.

CRM Technology Foundation APIs

The class oracle.apps.jtf.base.Logger delegates log calls to the AppsLog class. The following are the main logging APIs provided:

Logger.out(String message, int severity, Class module);

Use this API to log your message. The message length can be up to 4000 characters. For example:

```
public class MyClass {
…
public boolean myAPI() {
  …
  if(Logger.isEnabled(Logger.STATEMENT)) // Important check for Pe
rformance!
    Logger.out("My message", Logger.STATEMENT, MyClass.class);
}
}
```

Logger.out(String message, int severity, Object module);

In situations where the "Class" is not available (such as when writing a JSP), you can use this API and pass in a String. The message length can be up to 4,000 characters. For example:

```
<%  if(Logger.isEnabled(Logger.ERROR))  // Important check for Pe
rformance!
      Logger.out("In JSP land use the JSP Name", Logger.ERROR,
"jtf.html.jtftest.jsp"); %>
```

Logger.out(Exception e, Class module);

Use this API to log an exception. If the "Class" is not available, you can pass in the String object. If the exception length is greater than 4,000 characters, then the exception is split and logged in multiple rows. By default, all exceptions are logged at severity EXCEPTION. If you would like to log an exception at a different severity, you can use the corresponding APIs that take the severity as one of the arguments.

```
For example:
Logger.out(Exception e, int severity, Class module);
```

> **Note:** Do not specify integer values (1, 2, 3, etc.) in your calls to Logger APIs. Instead, refer to the severity level by the appropriate name:
>
> Logger.STATEMENT
>
> Logger.PROCEDURE
>
> Logger.EVENT
>
> Logger.EXCEPTION
>
> Logger.ERROR

Logger.UNEXPECTED

How to Log from PL/SQL

PL/SQL APIs are a part of the FND_LOG Package. These APIs assume that appropriate application user session initialization APIs (for example, FND_GLOBAL.INITIALIZE(..)) have already been invoked for setting up the user session properties on the database session. These application user session properties (UserId, RespId, AppId, SessionId) are internally needed for the Log APIs. In general, all Oracle Application frameworks invoke these session initialization APIs for you.

To log plain text messages, use FND_LOG.STRING(..),.

API Description

```
PACKAGE FND_LOG IS
   LEVEL_UNEXPECTED CONSTANT NUMBER  := 6;
   LEVEL_ERROR      CONSTANT NUMBER  := 5;
   LEVEL_EXCEPTION  CONSTANT NUMBER  := 4;
   LEVEL_EVENT      CONSTANT NUMBER  := 3;
   LEVEL_PROCEDURE  CONSTANT NUMBER  := 2;
   LEVEL_STATEMENT  CONSTANT NUMBER  := 1;

  /*
   **  Writes the message to the log file for the specified
   **  level and module
   **  if logging is enabled for this level and module
   */
   PROCEDURE STRING(LOG_LEVEL IN NUMBER,
                    MODULE    IN VARCHAR2,
                    MESSAGE   IN VARCHAR2);

   /*
   **  Writes a message to the log file if this level and module
   **  are enabled.
   **  The message gets set previously with FND_MESSAGE.SET_NAME,
   **  SET_TOKEN, etc.
   **  The message is popped off the message dictionary stack,
   **  if POP_MESSAGE is TRUE.
   **  Pass FALSE for POP_MESSAGE if the message will also be
   **  displayed to the user later.
   **  Example usage:
   **  FND_MESSAGE.SET_NAME(...);    -- Set message
   **  FND_MESSAGE.SET_TOKEN(...);   -- Set token in message
   **  FND_LOG.MESSAGE(..., FALSE);  -- Log message
   **  FND_MESSAGE.RAISE_ERROR;      -- Display message
   */
   PROCEDURE MESSAGE(LOG_LEVEL   IN NUMBER,
                     MODULE      IN VARCHAR2,
                     POP_MESSAGE IN BOOLEAN DEFAULT NULL);

   /*
   ** Tests whether logging is enabled for this level and module,
   ** to avoid the performance penalty of building long debug
   ** message strings unnecessarily.
   */
   FUNCTION TEST(LOG_LEVEL IN NUMBER, MODULE IN VARCHAR2)
RETURN BOOLEAN;
```

Example

Assuming Oracle Applications Object Library session initialization has occurred and logging is enabled, the following calls would log a message:

```
begin

  /* Here is where you would call a routine that logs messages */
  /* Important performance check, see if logging is enabled */
  if( FND_LOG.LEVEL_PROCEDURE >= FND_LOG.G_CURRENT_RUNTIME_LEVEL )
 then
    FND_LOG.STRING(FND_LOG.LEVEL_PROCEDURE,
        'fnd.plsql.MYSTUFF.FUNCTIONA.begin', 'Hello, world!' );
  end if;
/
```

The FND_LOG.G_CURRENT_RUNTIME_LEVEL global variable allows callers to avoid a function call if a log message is not for the current level. It is automatically populated by the FND_LOG_REPOSITORY package.

```
if( FND_LOG.LEVEL_EXCEPTION >= FND_LOG.G_CURRENT_RUNTIME_LEVEL )
then
      dbg_msg := create_lengthy_debug_message(...);
      FND_LOG.STRING(FND_LOG.LEVEL_EXCEPTION,
           'fnd.form.ABCDEFGH.PACKAGEA.FUNCTIONB.firstlabel', dbg_
msg);
end if;
```

For Forms Client PL/SQL, the APIs are the same. However to check if logging is enabled, you should call `FND_LOG.TEST(..)`.

For example, when logging Message Dictionary Messages:

```
if( FND_LOG.LEVEL_UNEXPECTED >=
    FND_LOG.G_CURRENT_RUNTIME_LEVEL) then
    FND_MESSAGE.SET_NAME('FND', 'LOGIN_ERROR'); -- Seeded Message
    -- Runtime Information
    FND_MESSAGE.SET_TOKEN('ERRNO', sqlcode);
    FND_MESSAGE.SET_TOKEN('REASON', sqlerrm);
    FND_LOG.MESSAGE(FND_LOG.LEVEL_UNEXPECTED, 'fnd.plsql.Login.val
idate', TRUE);
end if;
```

How to Log from C

Use the following APIs to log from C:

```
#define  AFLOG_UNEXPECTED  6
#define  AFLOG_ERROR       5
#define  AFLOG_EXCEPTION   4
#define  AFLOG_EVENT       3
#define  AFLOG_PROCEDURE   2
#define  AFLOG_STATEMENT   1

/*
** Writes a message to the log file if this level and module is
** enabled
*/
void aflogstr(/*_ sb4 level, text *module, text* message _*/);

/*
** Writes a message to the log file if this level and module is
** enabled.
** If pop_message=TRUE, the message is popped off the message
** Dictionary stack where it was set with afdstring() afdtoken(),
** etc. The stack is not cleared (so messages below will still be
** there in any case).
*/
void aflogmsg(/*_ sb4 level, text *module, boolean pop_message _*/
);

/*
** Tests whether logging is enabled for this level and module, to
** avoid the performance penalty of building long debug message
** strings
*/
boolean aflogtest(/*_ sb4 level, text *module _*/);

/*
** Internal
** This routine initializes the logging system from the profiles.
** It will also set up the current session and username in its sta
te */
void afloginit();
```

How to Log in Concurrent Programs

Debug and Error Logging

Use a CP Request Log only for messages intended for end users. Log debug information and error details (intended for system administrators and support personnel) to FND_LOG.

PL/SQL, Java, or C code that could be invoked by both CPs and application code should only use Oracle Applications Object Library (FND) Log APIs. If needed, the wrapper CP should perform appropriate batching and logging to the Request Log for progress reporting purposes.

For message correlation, in Release 11.5.10 and later, CP Request Log APIs log messages to both the Request Log and FND Log at severity EVENT (only if logging is enabled at EVENT or a lower level).

In Java CPs, use AppsLog for debug and error logging. The AppsLog instance can be obtained from the CpContext Object by calling getLog().

Request Log

> **Caution:** Do not use the Request Log for debug messages or internal error messages that are oriented to system administrators and/or Oracle Support. Such messages should only be logged to FND_LOG.

The Request Log is the end user UI for concurrent programs (CPs). When writing CP code, only translatable, end user-oriented messages should be logged to the Request Log.

For example, if an end user inputs a bad parameter to the CP, then log an error message to the Request Log so the end user can take corrective action. A code sample follows:

```
-- Seeded Message for End-User
FND_MESSAGE.SET_NAME('FND', 'INVALID_PARAMETER');
-- Runtime Parameter Information
FND_MESSAGE.SET_TOKEN('PARAM_NAME', pName);
FND_MESSAGE.SET_TOKEN('PARAM_VALUE', pValue);
-- Useful for Auto-Logging Errors
FND_MESSAGE.SET_MODULE('fnd.plsql.mypackage.myfuntionA');
fnd_file.put_line( FND_FILE.LOG, FND_MESSAGE.GET );
```

However, if the CP fails due to an internal software error, then the detailed failure message should be logged to FND_LOG. Additionally, a high-level generic message such as "Your request could not be completed due to an internal error"should also be logged to the Request Log to inform the end user of the error.

Output File

> **Caution:** Do not use the Output File for debug messages or internal error messages that are oriented to system administrators and/or Oracle Support. Such messages should only be logged to FND_LOG.

An output file is a formatted file generated by a CP that could be sent to a printer or viewed in a UI window. An invoice is an example of an output file, for example:

```
fnd_file.put_line( FND_FILE.OUTPUT, \******** XYZ Invoice *******
*' );
```

How to Raise System Alerts

Raise System Alerts to notify system administrators of serious problems or potentially serious problems. System Alerts are posted to the Oracle Applications Manager console, and are also sent to subscribed administrators through Workflow notifications. These messages should be used in cases where one of the following applies:

- The person who needs to take action is not the end user who encountered the problem.
- The problem is encountered by system processes, where there is no end user.

When a System Alert is posted, a variety of context information is automatically collected. This may include information about the end user, responsibility, product, component, OS process, database session, and so on. Oracle Applications Manager allows users to drill down from a System Alert message to view any collected context information, associated debug log messages, and other potentially relevant information.

Additionally, Oracle Applications Manager tracks multiple occurrences of the same alert message to prevent duplicate notifications from being sent.

All system alert messages must be defined in the Message Dictionary using the messages form under the system administration responsibility.

Raising a System Alert

- The message must be logged at the UNEXPECTED severity.
- The message must be an encoded Message Dictionary message.
- The message must have two attributes set in the Message Dictionary to facilitate notification routing:
  - Category: System, Product, Security, or User.
  - Severity: Critical, Error, or Warning.

PL/SQL Code Sample

```
...
Exception
  when others then
    if( FND_LOG.LEVEL_UNEXPECTED >=
        FND_LOG.G_CURRENT_RUNTIME_LEVEL) then
    -- To be alertable, seeded message must have
    -- Alert Category & Serverity defined
    FND_MESSAGE.SET_NAME('FND', 'LOGIN_ERROR'); -- Seeded Message
    -- Runtime Information
    FND_MESSAGE.SET_TOKEN('ERRNO', sqlcode);
    FND_MESSAGE.SET_TOKEN('REASON', sqlerrm);
    FND_LOG.MESSAGE(FND_LOG.LEVEL_UNEXPECTED, 'fnd.plsql.Login.val
idate', TRUE);
   end if;
...
```

Java Code Sample

```
if(alog.isEnabled(Log.UNEXPECTED)) {
    // To be alertable, seeded Message MUST have Alert
    // Category & Severity defined.
        Message Msg = new Message("FND", "LOGIN_ERROR");
    Msg.setToken("ERRNO", sqle.getErrorCode(), false);
    Msg.setToken("REASON", sqle.getMessage(), false);
     alog.write("fnd.security.LoginManager.authenticate", Msg, Log
.UNEXPECTED);
}
```

Guidelines for Defining System Alerts

- Make System Alert messages short and concise. System Alerts summarize problems and are used in reports and notifications, which in turn provide links to the related details.
- Do not include context information tokens in System Alert messages. For example, do not include the concurrent program name, Form name, time, routine, user, responsibility, etcetera in System Alert messages. Such context information is collected automatically by the logging APIs, and would be redundant in the System Alert message. Also, the alert message is used for filtering duplicate notifications. Including context information in the system alert message would defeat this filtering mechanism.
- You must set a value for the "Category" attribute. This attribute is used to categorize alerts and route notifications to the appropriate subscription. The valid values are as follows:
  - System

    Alert messages with the category "System" are typically routed to technical users such as the system administrators or DBAs who maintain the technology stack.
  - Product

    Alert messages with the category "Product" are typically routed to functional administrators or product super users who take care of product setup and maintenance.
  - Security

    Alert messages with the category "Security" are to alert administrators about E-Business Suite security issues.
  - User

    Alert messages with the category "User" are to alert administrators about issues reported by end users of the E-Business Suite.
- You must set a value for the "Severity" attribute. This attribute is used for sorting and filtering in Oracle Applications Manager. Also, users may subscribe to notifications for alert messages based on this attribute. The valid values are "Critical," "Error," and "Warning." Use "Critical" when a serious error completely impedes the progress of an important business process or affects a large user community. Use "Error" for less serious, more isolated errors. Use "Warning" when it is unclear whether the error has a negative impact on users or business processes.
- Refer to the online help provided in Oracle Applications Manager for more information about System Alerts.

A

PL/SQL Helper Packages

Overview

This section describes PL/SQL helper packages:

- JTF_DIAGNOSTIC_ADAPTUTIL - This package provides helper APIs to initialize and manipulate data structures used by PL/SQL diagnostic tests.
- JTF_DIAGNOSTIC_COREAPI - This package provides methods that can be used in formatting test reports (both HTML and plain text).

Related Topics

Package JTF_DIAGNOSTIC_ADAPTUTIL, page A-1

Package JTF_DIAGNOSTIC_COREAPI, page A-5

Package JTF_DIAGNOSTIC_ADAPTUTIL

This package provides helper APIs to initialize and manipulate data structures used by PL/SQL diagnostic tests.

Function initInputTable

Usage

initInputTable RETURN JTF_DIAG_INPUTTBL

Returns

Returns an initialized JTF_DIAG_INPUTTBL object.

Function initReportClob

Usage

initReportClob RETURN CLOB

Returns

Returns an initialized CLOB object.

Function compareResults

Usage

compareResults(*oper* IN VARCHAR2, *arg1* IN VARCHAR2, *arg2* IN VARCHAR2) RETURN BOOLEAN

Arguments

This procedure takes a three arguments:

- *oper* - The operand of the operation that is to be performed, i.e., ">", "<", or "=".
- *arg1* - The expected String value .
- *arg2* – The string value that is to be tested.

For example, passing in **'=' 'string1' 'StRiNg'** would evaluate to true, as the two strings match. Comparison is not case-sensitive.

Note: These functions are included in the utility package to help you implement your test case. They are by no means the only way to compare results within the PL/SQL diagnostic test template. For example:

```
IF compareResults('=','STR1','STR2')  THEN
```

is logically the same as:

```
IF ('STR1' = 'STR2') THEN
```

Either can be used while writing test cases.

Function compareResults

Usage

compareResults(*oper* IN VARCHAR2, *arg1* IN INTEGER, *arg2* IN INTEGER) RETURN BOOLEAN

Arguments

This procedure takes three arguments:

- *oper* - The operator of the operation that is to be performed , i.e., ">", "<", or "=".
- *arg1* - The expected value.
- *arg2* - The value that is to be tested.

That is, passing in **> 50 1** would evaluate to true, as **50** is greater than **1**, and so on.

compareResults('>',1,50) would evaluate to false, as **1** is less than **50**.

Note: These functions are included in the utility package to help you implement your test case. They are by no means the only way to compare results within the adapter. For example:

```
IF compareResults('>',5000,10)  THEN
```

is logically the same as:

```
IF (5000 > 10) THEN
```

and both return BOOLEAN values.

Procedure constructReport

Usage

constructReport(*status* IN VARCHAR2, *errStr* IN VARCHAR2, *fixInfo* IN VARCHAR2, *isFatal* IN VARCHAR2) RETURN JTF_DIAG_REPORT

Parameters

- *status* – The result of the test: "SUCCESS", "WARNING", or "FAILURE".
- *errStr* - The error that has been populated by the user. It could be SQLERRM or a user-defined error message, but must be under 4000 characters in length.
- *fixInfo* – A string to help the user to fix the associated problem. It must be under 4000 characters in length.
- *isFatal* - Either TRUE or FALSE (string representations are not Boolean values).

Procedure getInputValue

getInputValue(*argName* IN VARCHAR2, *inputs* IN JTF_DIAG_INPUTTBL) RETURN VARCHAR2

Parameters

- *argname* - The name of the variable you want retrieved.
- *inputs* – A JTF_DIAG_INPUTTBL object which is where the associated value is to be extracted from.

Procedure addInput

Usage

addInput(*inputs* IN JTF_DIAG_INPUTTBL, *var* IN VARCHAR2, *val* IN VARCHAR2) RETURN JTF_DIAG_INPUTTBL

Parameters

- *inputs* - A JTF_DIAG_INPUTTBL object which is a table of JTF_DIAG_INPUTS This object breaks down into two varchar2 objects representing the variable name and a second varchar2 representing the value. The *inputs* parameter must be initialized and passed into each additional *addInput* call to accumulate the variables as the *inputs* variable gets appended to with the name,value and is then returned.
- *var* – The name of the variable to add (VARCHAR2).
- *val* – The associated value of the variable name passed in (VARCHAR2).

Returns

This function creates a new JTF_DIAG_INPUTS object from the variable and value passed in and returns this added pairing into the caller function as part of the JTF_DIAG_INPUTTBL object. As this method is overloaded and no "*showValue*" is

passed in this instance, this field is set as TRUE by default for this call. That is, the value field will be visible on the UI layer.

Function addInput

Usage

addInput (*inputs* IN JTF_DIAG_INPUTTBL, *var* IN VARCHAR2, *val* IN VARCHAR2, *showValue* IN BOOLEAN) RETURN JTF_DIAG_INPUTTBL

Parameters

- *inputs* - A JTF_DIAG_INPUTTBL object which is a table of JTF_DIAG_INPUTS. This object breaks down into two varchar2 objects representing the variable name and a second varchar2 representing the value. The *inputs* parameter must be initialized and passed into each additional *addInput* call to accumulate the variables as the *inputs* variable gets appended to with the name,value and is then returned.
- *var* – The name of the variable to add (VARCHAR2).
- *val* – The associated value of the variable name passed in (VARCHAR2).
- *showValue* – A Boolean value to indicate if field is confidential on the UI.

Returns

This function creates a new JTF_DIAG_INPUTS object from the variable and value passed in and returns this added pairing into the caller function as part of the JTF_DIAG_INPUTTBL object. The *showValue* parameter can either be set to TRUE or FALSE. If value is true then the value field is visible on the UI. If the value is false, then the value field is confidential on the UI and will be displayed as a hidden field by a series of asterisks in the value's place.

Procedure setUp Vars

Usage

setUpVars(*reportClob* OUT CLOB)

This procedure is deprecated. See **setUpVars** below.

Procedure setUp Vars

Usage

setUpVars

Replaces setUpVars(CLOB).

This takes no arguments. This procedure initializes global variables for the current session. For example:

- The report CLOB is initialized for the session.
- The global flag (*b_html_on*) indicating that the report is to be written in HTML is reset to false (*b_html_on)*
- The global flag is set to true when '@html' is the first word written to the CLOB.

Procedure addStringToReport

Usage

addStringToReport (*reportClob* IN OUT CLOB, *reportStr* IN LONG)

This procedure is deprecated. See **addStringToReport (..)** below.

Procedure addStringToReport

Usage

addStringToReport (*reportStr* IN LONG)

Replaces *addStringToReport (CLOB,LONG).* It takes a LONG representation of the report string and appends the string onto the end of the current report CLOB. You are responsible for adding any string formatting, such as new lines.

Function addOutput

Usage

FUNCTION addOutput(outputs IN JTF_DIAG_OUTPUTTBL,var IN VARCHAR2,val IN VARCHAR2) RETURN JTF_DIAG_OUTPUTTBL;

Function initOutputTable

Usage

FUNCTION initOutputTable RETURN JTF_DIAG_OUTPUTTBL;

Function addDependency

Usage

FUNCTION addDependency(dependencies IN JTF_DIAG_DEPENDTBL, val IN VARCHAR2) RETURN JTF_DIAG_DEPENDTBL;

Function initDependencyTable

Usage

FUNCTION initDependencyTable RETURN JTF_DIAG_DEPENDTBL;

Package JTF_DIAGNOSTIC_COREAPI

This package provides methods that can be used in formatting test reports (both HTML and plain text).

Procedure Line_Out

Usage

Line_Out ('String')

Parameters

Any text string.

Output

Writes the text to the report CLOB. This procedure is similar to the **addStringToReport** procedure in the JTF_DIAGNOSTIC_ADAPTUTIL package.

Example

```
begin

    JTF_DIAGNOSTIC_COREAPI.Line_Out('Run Gather Schema Statistics'
);
end;
```

Procedure Insert_Style_Sheet

Usage

Insert_Style_Sheet

Output

Inserts a style sheet into the output. This API is not normally needed, as the style sheet is automatically inserted with the header.

Procedure ActionErrorPrint

Usage

ActionErrorPrint ('String');

Parameters

Any text string.

Output

Displays the text string with the word ACTION prior to the string.

Example

```
begin
    ActionErrorPrint('Run Gather Schema Statistics');
end;
```

Procedure ActionPrint

Usage

ActionPrint ('String');

Parameters

Any text string.

Output

Displays the text string.

Example

```
begin
    ActionPrint('Run Gather Schema Statistics');
end;
```

Procedure ActionWarningPrint

Usage

ActionWarningPrint ('String');

Parameters

Any text string.

Output

Displays the text string in a warning format.

Example

```
begin
    ActionWarningPrint('Run Gather Schema Statistics');
end;
```

Procedure WarningPrint

Usage

WarningPrint ('String');

Parameters

Any text string.

Output

Displays the text string in warning format.

Example

```
begin
WarningPrint('Statistics are not up to date');
end;
```

Procedure ActionErrorLink

Usage

ActionErrorLink ('Pre_String','Note_Number','Post_String');

ActionErrorLink ('Pre_String','URL','Link_Text', 'Post_String')

Parameters

- *Pre_String* - The text to appear prior to the link
- *Note_Number* - The number of the Oracle*MetaLink* note being linked to.
- *URL* - Any valid URL.

- *Link_Text* - Text for the link to the URL.
- *Post_String* - Text to appear after the link.

Output

This API displays the pre-link string, the link (as specified either by the note number or by the URL and link text), and the post-link string all in the format of an Error Action. It outputs HTML only.

Example

```
begin
ActionErrorLink('For clarification see note', 112233.1, 'which pro
vides more information on the subject');
ActionErrorLink('For clarification see the', 'http://someurl.us.co
m/somepage.html','Development Homepage', 'which provides more info
rmation on the subject');
end;
```

Procedure ActionWarningLink

Usage

ActionWarningLink ('Pre_String','Note_Number','Post_String');

ActionWarningLink ('Pre_String','URL','Link_Text', 'Post_String');

Parameters

- *Pre_String* - The text to appear prior to the link.
- *Note_Number* - The number of the Oracle*MetaLink* note being linked to.
- *URL* - Any valid URL.
- *Link_Text* - The text for the link to the URL.
- *Post_String* - The text to appear after the link.

Output

This API displays the pre-link string, the link (as specified either by the note number or by the URL and link text), and the post-link string all in the format of an Warning Action. It outputs HTML only.

Example

```
begin
ActionWarningLink('For clarification see note', 112233.1, 'which p
rovides more information on the subject');
ActionWarningLink('For clarification see the', 'http://someurl.us.
com/somepage.html','Development Homepage', 'which provides more in
formation on the subject');
end;
```

Procedure ErrorPrint

Usage

ErrorPrint ('String');

Parameters

Any text string.

Output

Displays the text string.

Example

```
begin
    ErrorPrint('Statistics have not been run');
end;
```

Procedure Show_Table_Header

This is a private text-only procedure used by Display_SQL to display the headers.

Procedure SectionPrint

Usage

SectionPrint ('String');

Parameters

Any text string.

Example

```
begin
SectionPrint('Checking OE Parameters');
end;
```

Procedure Tab0Print

Usage

Tab0Print ('String');

Parameters

Any text string.

Output

Displays the text string without any indentation.

Example

```
begin
    Tab0Print('Layer 0');
end;
```

Procedure Tab1Print

Usage

Tab1Print ('String');

Parameters

Any text string.

Output

Displays the text string with a 0.25 inch indentation.

Example

```
begin
    Tab1Print('Layer 1');
end;
```

Procedure Tab2Print

Usage

Tab2Print ('String');

Parameters

Any text string.

Output

Displays the text string with a 0.5 inch indentation.

Example

```
begin
    Tab2Print('Layer 2');
end;
```

Procedure Tab3Print

Usage

Tab3Print ('String');

Parameters

Any text string.

Output

Displays the text string with a 0.75 inch indentation.

Example

```
begin
    Tab3Print('Layer 3');
end;
```

Procedure BRPrint

Usage

BRPrint;

Output

Displays a blank line.

Example

```
begin
    Tab3Print('Layer 3');
    BRPrint;
    Tab3Print('Layer 4');
end;
```

Procedure CheckFinPeriod

Usage

CheckFinPeriod ('Set of Books ID','Application ID');

Parameters

- *Set of Books ID* - The ID for the set of books.
- *Application ID* - The ID of the application whose periods are being checked.

Output

This API lists the number of defined and open periods and indicates the latest period. It produces warnings if no periods are open or if the current date is not in an open period.

Example

```
CheckFinPeriod(62, 222); -- Check open periods for AR SOB 62
CheckFinPeriod(202, 201); -- Check open periods for PO SOB 202
```

Procedure CheckKeyFlexfield

Usage

CheckKeyFlexfield ('Key Flexfield Code','Flexfield Structure ID','Print Header');

Parameters

- *Key Flexfield Code* - The code of the Key Flexfield to be displayed. For example, use **GL#** for the Accounting Flexfield.
- *Flexfield Structure ID* - The id_flex_num of the specific structure of the Key Flexfield whose details are to be displayed. If null (the default), the API prints the details of all structures.
- *Print Header* - A Booleanoperator (true or false) that indicates whether the output should print a heading before outputting the details of the Key Flexfield. The default is "true".

Returns

If a value is provided for the flexfield structure ID, this function returns an array of character strings with the following structure:

1. Name of the flexfield
2. Enabled flag
3. Frozen flag
4. Dynamic insert flag
5. Cross-validation allowed flag
6. Number of enabled segments defined
7. Number of enabled segments with value sets
8. "Y" if any segment has security, otherwise "N"

If no value is passed to the parameter, the function returns an array with null values.

Output

Displays important information about the flexfield, its structure, and the individual flexfield segments it contains.

Example

```
declare
flexarray V2T;
begin
    CheckKeyFlexfield('GL#', 50577, true);
    CheckKeyFlexfield('MSTK', null, false);
    flexarray := CheckKeyFlexfield('GL#', 12345, false);
end;
```

Procedure CheckProfile

Usage

CheckProfile ('Profile Name', UserID, ResponsibilityID, ApplicationID, 'Default Value', Indent Level);

Parameters

- *Profile Name* - The system name of the profile option being checked.
- *User ID* - The identifier of the Oracle Applications user for which the profile option is to be checked.
- *Responsibility ID* - The identifier of the responsibility for which the profile option is to be checked.
- *Default Value* - The value used as a default if the profile option is not set by the user. The default is NULL.
- *Indent Level* - The number of tabs (0,1,2,3) that the output should be indented. The default is 0.

Returns

If called as a function, the return value will be one of the following:

- The value of the profile option, if set
- "DOESNOTEXIST" if the profile option does not exist
- "DISABLED" if the profile option has been end-dated
- Null if the profile option is not set

Output

If the profile has been set, this API outputs the profile's current setting. If not set and a default value exists, the API displays a warning which indicates that the default value will be used and what that default value is. If the profile has not been set and no default value is supplied, the API displays an error which indicates that the profile option should be set. The output will be indented according to the Indent Level parameter supplied. If the profile option does not exist or is disabled, then the API has no output.

Example

```
declare
    profile_val fnd_profile_option_values.profile_option_value%typ
e;
begin
    profile_val := CheckProfile('PA_SELECTIVE_FLEX_SEG',g_user_id,

                                g_resp_id, g_appl_id, null, 1);
    CheckProfile('PA_DEBUG_MODE',g_user_id, g_resp_id, g_appl_id);
    CheckProfile('PA_DEBUG_MODE',g_user_id, g_resp_id, g_appl_id,'
Y',2);
end;
```

Function Column_Exists

Usage

Column_Exists ('Table Name','Column Name');

Parameters

- *Table Name* - The name of the table that contains the column being checked.
- *Column Name* - The name of the column being checked.

Returns

Returns "Y" if the column exists in the table, "N" if it does not.

Example

```
declare
    sqltxt varchar2(1000);
begin
    if Column_Exists('PA_IMPLEMENTATIONS_ALL','UTIL_SUM_FLAG') = '
Y'
then;
    sqltxt := sqltxt||' and i.util_sum_flag is not null';
  end if;
end;
```

Procedure Begin_Pre

Usage

Begin_Pre;

Output

Allows the following output (HTML output only) to be preformatted.

Example

```
begin
    Begin_Pre;
end;
```

Procedure End_Pre

Usage

End_Pre;

Output

Closes the **Begin_Pre** procedure. For HTML output only.

```
begin
    End_Pre;
end;
```

Procedure Display_SQL

Usage

Display_SQL ('SQL statement', 'disp_lengths_tbl', 'headers_tbl', 'feedback', 'max rows');

Output

For text output.

Function Display_SQL

Usage

For HTML output:

a_number := Display_SQL('SQL Statement','Name for Header','Long Flag', 'Feedback', 'Max Rows');

For text output:

a_number := Display_SQL('SQL Statement', 'disp_lengths_tbl', 'headers_tbl', 'Feedback', 'Max Rows');

Parameters

- *SQL Statement* - A valid SQL select statement.
- *Name for Header* - A text string to serve as a heading for the output.
- *Long Flag* - "Y" or "N". If set to "N", then the API will not output any LONG columns. The default is "Y".
- *Feedback* - "Y" or "N". Defines whether to indicate the number of rows selected automatically in the output. The default is "Y".
- *Max Rows* - Limits the number of output rows to this number. A value of null or zero indicates there can be an unlimited number of output rows. The default is NULL.
- *disp_lengths_tbl* - A table of type "lengths" indicating the display length for each of the columns in the select. A value must be supplied for each column. If the value is null, the length of the header will be used.
- *headers_tb*l - A table of type "headers" indicating the column heading for each of the columns in the select. If an individual element of this parameter is null, or if this parameter is not provided (it is not required), the heading will be the column alias and the column name.

Returns

This function returns the number of rows selected. If there is an error, then the function returns -1.

Output

Displays an HTML table.

Example

```
declare
    num_rows number;
begin
    num_rows := Display_SQL('select * from ar_system_parameters_al
l', 'AR Parameters', 'Y', 'N',null);
    num_rows := Display_SQL('select * from pa_implementations_all'
, 'PA Implementation Options');
end;
```

Function Run_SQL

Usage

For HTML-only APIs:

a_number := Run_SQL('Heading', 'SQL statement');

a_number := Run_SQL('Heading', 'SQL statement', 'Feedback');

a_number := Run_SQL('Heading', 'SQL statement', 'Max Rows');

a_number := Run_SQL('Heading', 'SQL statement', 'Feedback', 'Max Rows');

For text-only APIs:

a_number := Run_SQL('Heading', 'SQL statement','disp_lengths_tbl', 'col_headers_tbl');

a_number := Run_SQL('Heading', 'SQL statement','disp_lengths_tbl','col_headers_tbl','Feedback');

a_number := Run_SQL('Heading', 'SQL statement','disp_lengths_tbl', 'col_headers_tbl', 'Max Rows');

a_number := Run_SQL('Heading', 'SQL statement','disp_lengths_tbl', 'col_headers_tbl', 'Feedback', 'Max Rows');

Parameters

- *Heading* - A text string to serve as a heading for the output.
- *SQL Statement* - Any valid SQL select statement.
- *Feedback* - "Y" or "N". Indicates whether to automatically print the number of rows returned. The default is "Y".
- *Max Rows* - Limits the number of output rows to this number. A value of null or zero indicates there can be an unlimited number of output rows. The default is NULL.
- *disp_lengths_tbl* - A table of type "lengths" indicating the display length for each of the columns in the select. A value must be supplied for each column. If the value is null, the length of the header will be used.
- *headers_tbl* - A table of type "headers" indicating the column heading for each of the columns in the select. If an individual element of this parameter is null, or if this parameter is not provided (it is not required), the heading will be the column alias and the column name.

Returns

This function returns the number of rows selected. If there is an error, then the function returns -1.

Output

Displays the SQL statement's output as an HTML table.

Example

```
declare
    num_rows number;
begin
    num_rows := Run_SQL('AR Parameters', 'select * from ar_system_
parameters_all');
end;
```

Function Run_SQL

Usage

For HTML-only APIs:

Run_SQL('Heading', 'SQL statement');

Run_SQL('Heading', 'SQL statement', 'Feedback');

Run_SQL('Heading', 'SQL statement', 'Max Rows');

Run_SQL('Heading', 'SQL statement', 'Feedback', 'Max Rows');

For text-only APIs:

Run_SQL('Heading', 'SQL statement', 'disp_lengths_tbl', 'col_headers_tbl');

Run_SQL('Heading', 'SQL statement', 'disp_lengths_tbl','col_headers_tbl','feedback');

Run_SQL('Heading', 'SQL statement', 'disp_lengths_tbl','col_headers_tbl','max rows');

Run_SQL('Heading', 'SQL statement', 'disp_lengths_tbl','col_headers_tbl','feedback', 'max rows');

Parameters

- *SQL statement* - A valid SQL select statement.
- *Heading* - A text string to be a heading for the output.
- *disp_lengths_tbl* - A table of type "lengths" that indicates the display length for each of the columns in the select. A value must be supplied for each column, even if that value is null. If the value is null, the length of the header will be used.
- *col_headers_tbl* - A table of type "headers" that indicates the column heading for each of the columns in the select. If an individual element of this parameter is null, or if this parameter is not provided (it is not required), the heading will be the column alias and the column name.

Output

Displays the SQL statement's output as an HTML table.

Example

```
begin
    Run_SQL('AR Parameters', 'select * from ar_system_parameters_a
ll');
end;
```

Function Compare_Pkg_Version

Usage

Compare_Pkg_Version ('package_name','obj_type','obj_owner', 'outversvar', 'reference_version');

Compare_Pkg_Version ('package_name','obj_type', 'outversvar', 'reference_version');

Parameters

- *package_name* - The name of the package whose version is being checked.
- *obj_type* - Either "BODY" or "SPEC", to determine which piece to check.
- *obj_owner* - The owner of the package being checked. If null or not supplied, the default value is "APPS".

- *outversvar* - A text-out variable to hold the actual package version of the package as returned from the database.
- *reference_version* - A string containing the version to which the package version should be compared. Uses the format ###.### -- for example, 115.119 (rather than 11.5.119).

Returns

- "greater" if the version of the object is greater than the reference
- "less" if the version of the object is less than the reference
- "equal" if the version of the object is equal to the reference
- "null" if either the reference or database version is null

Output

Text only.

Example

```
declare
Comparison_Var varchar2(8);
Package_Version varchar2(10);
begin
Comparison_Var := Compare_Pkg_Version('PA_UTILS2','BODY','APPS', P
ackage_Version, '115.13');
Comparison_Var := Compare_Pkg_Version('PA_UTILS2','BODY', Package_
Version, '115.13');
end;
```

Procedure Show_Responsibilities

Usage

Show_Responsibilities ('username');

Parameters

- *username* = a valid Oracle Applications username (case insensitive)

Output

Text only.

Example

```
begin
    Show_Responsibilities('jdoe');
end;
```

Function Display_Table

Usage

Display_Table ('Table Name', 'Heading', 'Where Clause', 'Order By', 'Long Flag');

Parameters

- *Table Name* - A valid table or view.
- *Heading* - A text string to serve as the output heading.
- *Where Clause* - The where clause to apply to the table dump.
- *Order By* - The "order by" clause to apply to the table dump.
- *Long Flag* - "Y" or "N". If set to "N", then this will not output any LONG columns.

Output

Displays the output of the "select * from table" as an HTML table. This API only outputs HTML.

Example

```
begin

    Display_Table('AR_SYSTEM_PARAMETERS_ALL', 'AR Parameters', 'Wh
ere Org_id != -3113', 'order by org_id, set_of_books_id', 'N');
end;
```

Function Display_Table

Usage

a_number := Display_Table('Table Name', 'Heading', 'Where Clause', 'Order By', 'Long Flag');

Parameters

- *Table Name* - A valid table or view.
- *Heading* - A text string to serve as the output heading.
- *Where Clause* - The where clause to apply to the table dump.
- *Order By* - The "order by" clause to apply to the table dump.
- *Long Flag* - "Y" or "N". If set to "N", then this will not output any LONG columns.

Returns

The number of rows displayed.

Output

Displays the output of the "select * from table" as an HTML table. This API only outputs HTML.

Example

```
declare
  num_rows number;
begin
    num_rows := Display_Table('AR_SYSTEM_PARAMETERS_ALL', 'AR Para
meters', 'Where Org_id <>  -3113' , 'order by org_id, set_of_books
_id', 'N');
end;
```

Function Get_DB_Apps_Version

Usage

a_varchar := Get_DB_Apps_Version;

Returns

Returns the version of applications found in fnd_product_groups. It also sets the variable g_appl_version to "10.7","11.0", or "11.5" as appropriate.

Example

```
declare
    apps_ver   varchar2(20);
begin
      apps_ver := Get_DB_Apps_Version;
end;
```

Procedure Show_Header

Usage

Show_Header ('Note Number', 'Title');

Parameters

- *Note Number* - A valid Oracle*MetaLink* note number.
- *Title* - A text string to display next to the note link.

Output

Displays standard header information.

Example

```
begin
    Show_Header('139684.1', 'Oracle Applications Current Patchsets
 Comparison to applptch.txt');
end;
```

Procedure Show_Footer

Usage

Procedure Show_Footer ('Script Name','Header');

Output

Displays a standard footer.

Example

```
begin

   Show_Footer('AR Setup Script', '$Header: ARTrxInfo.sql 1.0 01/1
2/11 12:33:24 support $';
end;
```

Procedure Show_Link

Usage

Procedure Show_Link ('Note Number');

Output

Displays a link to an Oracle*MetaLink* note.

Example

```
begin
    Show_Link('139684.1');
end;
```

Procedure Show_Link

Usage

Show_Link('URL', 'Name');

Output

Displays a link to a URL using the name parameter value.

Example

```
begin
Show_Link('http://metalink.us.oracle.com', 'OracleMetaLink');
end;
```

Procedure Send_Email

Usage

Send_Email ('Sender', 'Recipient', 'Subject', 'Message', 'SMTP Host');

Output

Sends an e-mail message. There is no screen output.

Example

```
begin
Send_Email ('sender@company.com','recipient@oracle.com','This is a
 subject', 'This is a message body','gmsmtp01.oraclecorp.com');
end;
```

Function Get_Package_Version

Usage

a_varchar := Get_Package_Version ('Object Type', 'Schema', 'Package Name');

Returns

Returns the version of the package or specification.

Example

```
declare
    spec_ver varchar2(20);
    body_ver varchar2(20);
begin
    spec_ver := Get_Package_Version('PACKAGE','APPS','ARH_ADDR_PKG
');
    body_ver := Get_Package_Version('PACKAGE BODY','APPS','ARH_ADD
R_PKG');
end;
```

Function Get_Package_Spec

Usage

a_varchar := Get_Package_Spec('Package Name');

Returns

Returns the version of the package specification in the APPS schema.

Example

```
declare
    spec_ver   varchar2(20);
begin
    spec_ver := Get_Package_Spec('ARH_ADDR_PKG');
end;
```

Function Get_Package_Body

Usage

a_varchar := Get_Package_Body('Package Name');

Returns

Returns the version of the package body in the APPS schema.

Example

```
declare
    body_ver   varchar2(20);
begin
    body_ver := Get_Package_Body('ARH_ADDR_PKG');
end;
```

Procedure Display_Profiles

Usage

Display_Profiles (application ID, 'profile short name');

Output

Displays all profile settings for the application or profile in an HTML table.

Example

```
begin
    Display_Profiles(222,null);
    Display_Profiles(null, 'AR_ALLOW_OVERAPPLICATION_IN_LOCKBOX');
end;
```

Function Get_Profile_Option

Usage

a_varchar := Get_Profile_Option('Short Name');

Parameter

- *Short Name* - The short name of the profile option.

Returns

Returns the value of the profile option, based on the user. If Set_Client has not been run successfully, then it will return the site-level profile option value.

Example

```
declare
    prof_value varchar2(150);
begin
    prof_value := Get_Profile_Option('AR_ALLOW_OVERAPPLICATION_IN_
LOCKBOX')
end;
```

Procedure Set_Org

Usage

Set_Org (Org_ID);

Parameters

- *Org_ID* - The identifier of the organization to be set.

Output

None

Example

```
begin
    Set_Org(204);
end;
```

Procedure Set_Client

Usage

Set_Client(UserName, Responsibility_ID);

Set_Client(UserName, Responsibility_ID, Application_ID);

Set_Client(UserName, Responsibility_ID, Application_ID, SecurityGrp_ID);

This procedure validates the UserName, Responsibility_ID, and Application_ID parameters. If valid, it initializes the session, which results in the operating unit being set for the session as well. It also sets the global variables g_user_id, g_resp_id, g_appl_id, and g_org_id, which can then be used throughout the script.

Parameters

- *UserName* - The name of the Oracle Applications user.
- *Responsibility_ID* - A valid responsibility ID.
- *Application_ID* - A valid application ID. If no value is provided, an attempt will be made to obtain it from the responsibility ID.
- *SecurityGrp_ID* - A valid security group ID.

Example

```
begin
    Set_Client('JOEUSER',50719, 222);
end;
```

Procedure Get_DB_Patch_List

Usage

a_string := Get_DB_Patch_List('Heading', 'Short Name', 'Bug Number', 'Start Date');

Parameters

- *Heading* - A text heading for the TABLE or TEXT outputs.
- *Short Name* - The short name of the Oracle Applications product.
- *Bug Number* - The bug number identifier.
- *Start Date* - The earliest applicable bug creation date.

Output

Displays an HTML table of patches that have been applied for the application since the start date.

Example

```
begin
    Get_DB_Patch_List(null, 'AD','%', '03-03-2002', 'SILENT');
end;
```

Function Get_RDBMS_Header

Usage

Get_RDBMS_Header;

Returns

The version of the database from v$version.

Example

```
declare
RDBMS_Ver := v$version.banner%type;
begin
RDBMS_Ver := Get_RDBMS_Header;
end;
```

Procedure Show_Invalids

Usage

Show_Invalids('Start String', 'Include Errors', 'Heading');

Parameters

- *Start String* - A string indicating the beginning of object names to be included. The underscore character (_) will be escaped in this string so that it does not act as a wild card character. For example, "PA_" will not match "PAY", even though it normally would in SQL*Plus.
- *Include Errors* - "Y" or "N". Indicates whether to search on and report the errors from ALL_ERRORS for each of the invalid objects found. The default is "N".
- *Heading* - An optional heading for the table. If null, the heading will be "Invalid Objects (Starting with 'XXX')" where XXX is the Start String parameter.

Output

This procedure outputs a list of invalid objects whose names starts with the Start String. For packages, procedures, and functions, file versions will be included. When requested, error messages associated with the object will be reported.

Example

```
Show_Invalids('GL_');
```

B

SQL Trace Options

SQL Trace Options

Oracle E–Business Suite Forms–based applications allow you to set up SQL Trace under the Help > Diagnostics menu. The trace options allow you to have server and background processes write information to associated trace files. When a process detects an internal error, it writes information about the error to its trace file. For more information on trace files, see the Oracle database documentation.

> **Note:** Enabling SQL Trace can have a severe performance impact. For more information, see the Oracle database documentation.

The following options are available:

- No Trace – turns trace off.
- Regular Trace – generates a regular SQL trace by performing the following statement:

```
ALTER SESSION SET SQL_TRACE = TRUE;
```

- Trace with Binds – writes bind variable values in the SQL trace file
- Trace with Waits – writes wait events in the SQL trace file
- Trace with Binds and Waits – writes both bind variable values and wait events in the SQL trace file
- Unlimited Trace File Size – allows an unlimited size for the trace file

Once SQL Trace is enabled using the Help >Diagnostics menu, the system enables trace for any form launched from the form in which trace was enabled. If trace is enabled while the Navigator is in focus, any subsequent form launched has trace enabled. When any subsequent forms are launched, the menu option indicates that trace is enabled.

A message is displayed at form startup indicating that trace is enabled.

Index

L

M

O

P

R

S

T