IN THE UNITED STATES DISTRICT COURT
FOR THE DISTRICT OF DELAWARE

ORACLE CORPORATION and ORACLE U.S.A. INC.,	)	
	)	C.A. No. 06-414-SLR
Plaintiffs/Counterclaim Defendants,	)	
	)	**JURY TRIAL DEMANDED**
v.	)	
	)	**PUBLIC VERSION**
EPICREALM LICENSING, LP,	)	
	)	
Defendant/Counterclaim Plaintiff.	)	

SECOND DECLARATION OF DR. DAVID FINKEL IN SUPPORT OF EPICREALM'S OPPOSITIONS TO ORACLE'S MOTIONS FOR SUMMARY JUDGMENT OF NONINFRINGEMENT AND INVALIDITY

VOLUME II of V

EXHIBITS 11 to 26

Richard L. Horwitz (#2246)
David E. Moore (#3983)
POTTER ANDERSON & CORROON LLP
Hercules Plaza, 6th Floor
1313 N. Market Street
Wilmington, DE 19899
Tel: (302) 984-6000
rhorwitz@potteranderson.com
dmoore@potteranderson.com

Attorneys for Defendant/Counterclaim Plaintiff epicRealm Licensing, LP

OF COUNSEL:

Harry J. Roper
George S. Bosy
Aaron A. Barlow
Patrick L. Patras
David R. Bennett
Paul D. Margolis
Benjamin J. Bradford
Emily C. Johnson
JENNER & BLOCK
330 N. Wabash Avenue
Chicago, IL 60611-7603
Tel: (312) 923-8305

Dated: August 21, 2008
Public Version Dated: August 29, 2008
880482 / 31393 / Oracle

IN THE UNITED STATES DISTRICT COURT
FOR THE DISTRICT OF DELAWARE

ORACLE CORPORATION and ORACLE U.S.A. INC.,	)	
Plaintiffs,	)	C.A. No. 06-cv-414 (SLR)
v.	)	
EPICREALM LICENSING, LP,	)	
Defendant.	)	

SECOND DECLARATION OF DR. DAVID FINKEL IN SUPPORT OF EPICREALM'S OPPOSITIONS TO ORACLE'S MOTIONS FOR SUMMARY JUDGMENT OF NONINFRINGEMENT AND INVALIDITY

I, David Finkel, declare as follows:

1. I am currently the Co-Director of the Computer and Communications Networks program at Worcester Polytechnic Institute. I have been a professor of Computer Science since 1991. I submit this declaration in support of EpicRealm's Oppositions to Oracle's Motions For Summary Judgment of Noninfringement and Invalidity.

2. I have been involved in studying, evaluating, and teaching about computer networks, network software, and the World Wide Web for over 15 years, and have authored numerous articles in the field. Additionally, I have taught undergraduate and graduate courses in the areas of computer networks and computer communications since 1989. Furthermore, I have taught several courses that specifically focus on Internet technology and the World Wide Web.

3. The Accused Oracle Application Server Products literally infringe Claim 1 of the '554 Patent under both parties' constructions. (*See* 2d Finkel Decl. Ex. 1 at 1-20.)

4. The Accused Oracle Application Server Products literally meet the "intercepting" limitation under either party's construction. (*See* 2d Finkel Decl. Exs. 1, 2.)

5. The "Web server" in all of the Accused Oracle Products processes some, but not all requests.

6. All software always goes through its "same iterative process," *i.e.*, it always follows its sequence of instructions. The software code does not change.

7. In Leaf, all requests are immediately forwarded on from the so-called "Web server" to transaction gateways; the "Web server" in Leaf could not process requests itself. Thus, a request could not be "intercepted" at the "Web server" in Leaf because there is no determination of whether the "Web server" or a "page server" should process the request—the

"Web server" in Leaf would never process a request itself. Therefore, Leaf did not disclose "intercepting."

8. The Accused Oracle Application Server Products literally meet the "releasing" limitation under either party's construction because the "Web server" (OHS) is released to process "other requests." (*See* 2d Finkel Decl. Exs. 1, 2.)

Redacted

Redacted

12. The Accused Oracle Application Server Products literally meet the "dispatcher" limitation under both its ordinary meaning and Oracle's construction. (*See* 2d Finkel Decl. Ex. 1, 2.)

13. Mod_oc4j meets Oracle's construction of "dispatcher" because it is a software program that determines which page server should process a dynamic Web page generation request.

14. Source code is written in modular fashion. For example, within one piece of software there can be one section of code that performs the "Web server" functionality and another section of code that performs the "dispatcher" functionality. Therefore, a request is routed from the "Web server" to the "dispatcher" when the request is routed from the code that performs the "Web server" functionality to the code that performs the "dispatcher" functionality.

15. The "Database Connector" is a Dynamically Linked Library (".dll") that runs in the same executable as the IIS 1.0 "Web server."

16. The Accused Oracle Database Products literally infringe claim 1 of the '554 Patent under both parties' constructions. (*See* 2d Finkel Decl. Ex. 3 at 1-21.)

Redacted

Redacted

22. Security attributes and privileges can come from a dynamic Web page request.

23. The Accused Oracle Web Cache Products literally infringe Claim 1 of the '554 Patent under both parties' constructions. (*See* 2d Finkel Decl. Ex. 5 at 1-19.)

24. The "Web server" (Web Cache) processes some but not all requests.

25. Web Cache is released to process other requests under either party's construction of "releasing." When Web Cache routes a request to a "page server," it is released to process

other requests under both parties' constructions in the same two ways—the same two ways that the "Web server" of the Accused Oracle Application Server Products are released.

Redacted

28. Merely having a "page server" process the request does not necessarily mean that the "releasing" limitation is met. For example, a system could be designed to have the "Web server" process a request in parallel with a "page server," *e.g.*, to help ensure a prompt response to a Web client even if the "page server" subsequently became slow or had a problem.

Redacted

31. The Accused Oracle Products also literally infringe the asserted claims that depend from Claim 1 of the '554 Patent. (*See* 2d Finkel Decl. Exs. 1, 3, 5.)

32. The Accused Oracle Products also literally infringe Claims 9 and 10 of the '554 Patent. (*See* 2d Finkel Decl. Exs. 1, 3, 5.)

33. The Accused Oracle Products also literally infringe Claim 11 of the '554 Patent. (*See* 2d Finkel Decl. Ex. 1, 3, 5.)

34. The Accused Oracle Products also literally infringe Claims 2 and 16 of the '335 Patent. (*See* 2d Finkel Decl. Ex. 2, 4, 6.)

35. Oracle provides various manuals to its customers instructing them how to use the Accused Oracle Products in an infringing manner.

36. The Accused Oracle Products infringe the asserted claims under the doctrine of equivalents. The Accused Oracle Products infringe because the differences between the Accused Oracle Products and the asserted claims are insubstantial from the perspective of a person of ordinary skill in the relevant art.

37. Nowhere in the specification does it discuss that "intercepting a request" requires that the Web server distinguish between dynamic and static content. Nor would one skilled in the art understand that the claims of the patents require such a distinction. Oracle Web Cache makes a determination of whether it will process a request itself (*e.g.*, satisfy the request with data stored in its cache) or it will send the request to an application server for processing.

38. The partitioned architecture of the claimed invention involves adding "page servers" which are capable of processing dynamic Web page requests. The "page servers" alleviate the processing burden placed on the "Web server" by processing some or all of the dynamic Web page requests. When a "page server" processes a request it "frees" or "releases" the "Web server" to process other requests because the "Web server" no longer has to expend resources processing that request. .

39. In the patented invention, the "dispatcher" can reside on a different machine than the "Web server" or the "dispatcher" can reside on the same machine as the "Web server." Regardless of where the "dispatcher" is located, it performs the same function. When the "dispatcher" resides on the same machine as the "Web server," some of the advantages of the invention will not be enjoyed. Namely, the Web server will not be able to offload the processing of the requests when a request is routed to the "dispatcher." The Web server, however, may still be able to offload the processing of a request when the request is routed from the "dispatcher"

residing on the Web server machine to a "page server." Thus, the purpose of the "dispatcher" is to "dispatch" requests, not to unburden the "Web server."

Redacted

41. The Oracle Web Agent does not make an informed selection of which page server should process the request. When the Oracle Web Agent determines which "page server" (Oracle7 Server) should process a request, it does not make a selection between Oracle7 Servers that are capable of processing the request. Rather, only one Oracle7 server could process the request. Oracle concedes that making an "informed ***selection***" requires, at least, choosing from more than one page server that is capable of processing the request. Thus, the Oracle Web Agent does not make an "informed selection."

42. The Oracle Web Agent also does not make a selection based on information indicating which page server can more efficiently process a request. Because only one Oracle7 server can process a given request, the information used by the Oracle Web Agent does not indicate the relative efficiencies of the Oracle7 servers. Instead, the information only indicates

which Oracle7 server to connect to and how to connect to that server. Thus, the Oracle Web Agent does not use information indicating which page server can more efficiently process a request in deciding which page sever should process a request.

43. The term "dynamic information" refers to information that can only be established during the execution of a program. Dynamic information does not refer to information that can be established prior to the execution of the program. Put another way, dynamic information is of a nature that can only be determined during runtime execution.

44. SQL*Net is remote data access software. Its fundamental purpose is to allow clients and servers to access Oracle databases from remote locations. SQL*Net is not concerned with the management of dynamic Web page requests. In fact, the term "Web page" is never used in the "Understanding SQL*Net 2.2" reference. Load balancing of connection requests has little to do with metric-based load balancing of dynamic Web page requests.

45. SQL*Net routes requests for database connections which are not the same as dynamic Web page requests as recited in all of the asserted claims, which are a type of "work requests." A request for a database connection attempts to establish a connection between a client and a server; the response to a request for a database connection will not be a dynamic Web page. By comparison, a dynamic Web page request typically results in a dynamic Web page in response and cannot establish a connection between a client and a database.

46. When a client is connecting to a multi-threaded database using SQL*Net 2.2 or SQL*Net 2.3, the request for a database connection sent from the client to the database is a wholly different request for a connection than the request originally received by SQL*Net.

47. One way of implementing the selection of the "least called" server is a round-robin algorithm.

48. Oracle 1.0 does not anticipate any of the asserted claims.

49. Claim 4 requires that the "page server" maintain a connection cache to said one or more data sources. Claim 5 requires that the processing step performed by the "page server" also includes the step of logging into one or more data sources. Oracle 1.0 does not anticipate Claims 4 and 5 of the '554 Patent for the additional reason that it cannot maintain a connection cache or log into one or more data sources. According to Oracle, the "page server" in Oracle 1.0 is the Oracle7 server, *i.e.* an Oracle database server. The data sources in Oracle 1.0 are the Oracle7 database. The Oracle7 server does not need to establish a connection to an Oracle7 database in order to retrieve data from the database because they are located on the same machine.

50. Oracle argues that Claim 5 is met because the "dispatcher" (the Oracle Web Agent) logs into the Oracle7 server. However, Claim 5 states that the "processing" step includes logging into one or more data sources and the Oracle Web Agent does not perform the "processing" step of Claim 1. Indeed, Oracle contends that the processing step is performed by the Oracle7 server, not the Oracle Web Agent.

51. Oracle 2.0 does not anticipate any of the asserted claims.

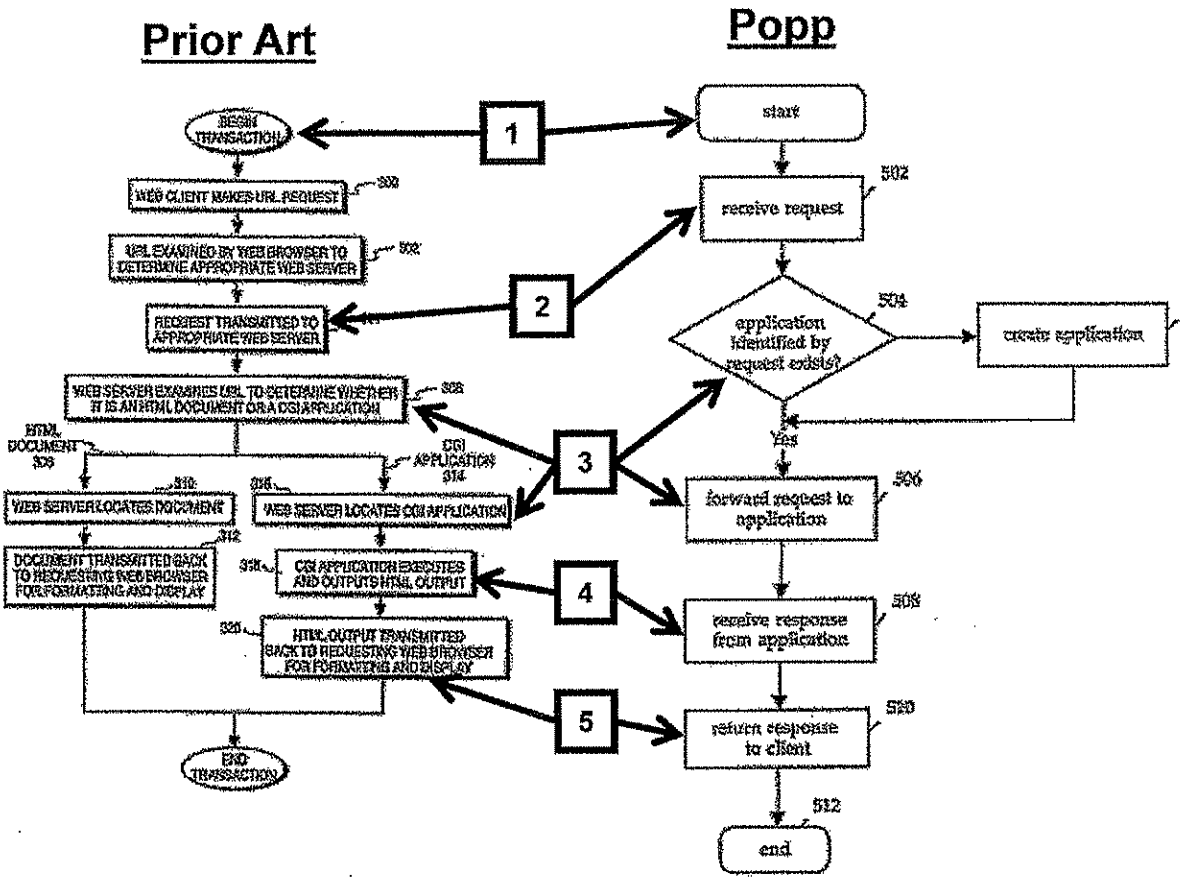
52. Oracle's proposed construction for page server is "page generating software, running on a processor separate from that of the Web server, that generates dynamic Web pages." However, Oracle's alleged "page server" in Oracle 2.0 is a WRBX that runs on the same machine (and therefore the same processor) as the Oracle 2.0 "Web server."

53. Oracle 2.0 does not anticipate Claim 4 for the additional reason that it does not disclose the "page server maintaining a connection cache to said one or more data sources." Under the ordinary meaning of "connection cache," as understood by one skilled in the art, maintaining a connection cache involved a cache of connections to a database that can be

checked out and reused by numerous processes. When a PL/SQL Agent stays connected to a database, there is neither a cache of connections nor can that connection be shared among various processes. Thus, it fails to meet the ordinary meaning of "connection cache."

54. Popp does not anticipate any of the asserted claims.

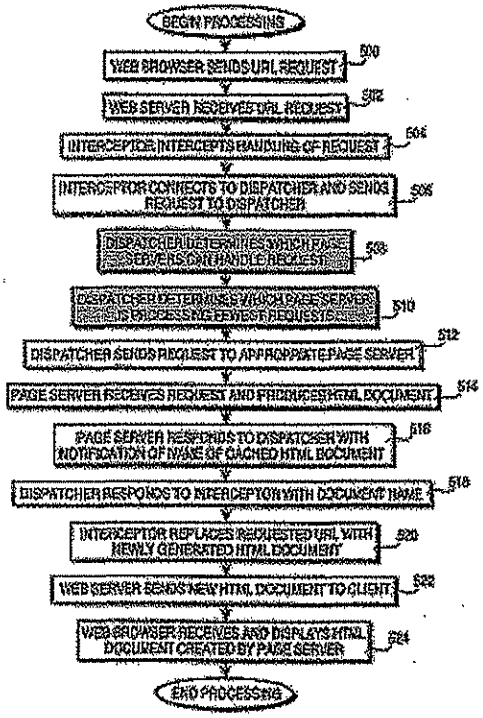
55. Below on the left is a flow chart describing how a request is processed in the prior art Web server disclosed by the '554 Patent and on the right is a flow chart from Popp describing how it processes a request:



nearly identical manner as the prior art disclosed by the patents in suit and considered by the

examiner. Specifically, Popp only determined if the application identified by a request existed, and if it did, forwarded the request to that application.

56. By comparison, in the flow diagram below depicting an embodiment of how the invention of the patents in suit processed a request, the "page server" is selected based on a two-step process (Steps 508 and 510 highlighted in gray below): (1) determine "which page servers can handle request;" and (2) determine "which page server is processing fewest requests" (*i.e.*, which "page server" can more efficiently process the request).



57. Popp's approach to Web site architecture is very similar to Oracle 1.0. In both cases, the primary concern was routing Web page requests to an application that could process the request. Neither Popp, nor Oracle 1.0, gave any consideration to determining which instance of an application "can *more efficiently* process the request." In fact, Oracle 1.0 and Popp are nearly identical to the prior art described in the '554 Patent specification.

58. By comparison, as shown in the embodiment illustrated in Figure 5 of the '554 Patent, the "page server" is selected using steps 508 ("dispatcher determines which page servers can handle request") and 510 ("dispatcher determines which page server is processing fewest requests"), *i.e.*, which "page server" can more efficiently process the request.

59. The CGI interface program in Popp (the alleged "dispatcher") does not make an "informed selection" of which page server should process a request. When the CGI interface program selects an application to process a request, it only determines which application can process a request because there is only one application that can process each request.

60. The CGI interface also does not make a selection based on information indicating which "page server" can more efficiently process a request. Because only one application can process a given request, the information used by the CGI interface does not indicate the relative efficiencies of the applications.

61. The information used by the CGI interface program in Popp is not "dynamic information maintained about page servers." The information relating to which application can process a request and on which server the application is executing is static information because it can be determined before runtime execution.

62. The CGI interface also does not make an "informed selection" because it does not use any information "to make a reasonable or accurate determination as to which [page server] could more efficiently process the request."

63. Dienst does not anticipate any of the asserted claims.

64. Dienst is not concerned with determining which "page server" can more efficiently process a request. Instead, it is concerned with locating the only "page server" that can process a request.

65. Dienst does not disclose "dispatching" for many of the same reasons that Oracle 1.0 and Popp do not disclose "dispatching." All these references are focused on locating the one and only "page server" that can process a given request.

66. The alleged "dispatcher" in Dienst does not make an "*informed selection*" of which page server should process a request. In Dienst, only one "page server" (Dienst Server) contains all the works for a specific publisher. Thus, when the alleged "dispatcher" selects a Dienst Server to process a request, it does not make a selection, but rather locates the only Dienst Server that contains that publisher's works.

67. The alleged "dispatcher" in Dienst also does not make a selection based on information indicating which "page server" can more efficiently process a request. The alleged "dispatcher" only uses information on which Dienst Server is capable of responding to a given request. Because there is only one Dienst Server that can process a request, there is no information on which Dienst Server can more efficiently process that request.

68. The information used by the CGI interface program is not dynamic information maintained about "page servers" indicating which "page server" can more efficiently process a request. Each Dienst Server downloads a listing of the locations of the other Dienst Servers so that it can search the other servers for works. This information is not dynamic information maintained about "page servers" indicating which "page server" can more efficiently process a request.

69. In my opinion, a person of ordinary skill in the art would have a Bachelor's degree in computer science or equivalent, with three to five years of technical experience in software implementation, or equivalent.

70. There is no "dispatching" under either party's proposed construction because the job of the alleged "dispatcher" is limited to locating the one "page server" that can process the given request.

71. Garland does not cure the "dispatching" deficiency in Oracle's invalidity arguments. The "Dispatcher Server" in Garland does not "examine" or "analyze" the request, as required by both parties' "dispatching" constructions. The "Dispatcher Server" also does not send the request to the selected server as required by both parties' "dispatching" constructions.

72. None of the asserted claims are obvious over the combinations of references argued by Oracle.

73. Oracle has not proved by clear and convincing evidence that Garland in combination with any of Oracle's four other references would have rendered obvious to one of ordinary skill in the art any of the twelve asserted claims at the time of the respective inventions.

74. Because Dienst is concerned with providing access to a distributed digital library over the Web, it does not disclose many of the limitations added by the dependent claims of the '554 Patent, which are focused on features to further improve the performance of a Web site. Oracle argues that it would be obvious to combine Dienst with the teachings of Oracle 1.0 or Oracle 2.0 to meet these claims. In addition, none of these references discloses "dispatching." Their combination would not change that fact. Not only does Oracle fail to set forth its basis for claiming that it would be obvious to combine these references, but the secondary considerations argue against such combinations.

75. Further, it would not be obvious to combine Dienst, a reference on providing access to a distributed digital library with Oracle 1.0 or Oracle 2.0. In fact, it would be impossible to use the "page servers" of Oracle 1.0 or Oracle 2.0 to implement Dienst.

76. The information used by JDBC to identify connections that are authenticated with the proper user credentials, user name and password and to identify connections that have the necessary striping can come from the request.

77. I am over the age of eighteen, and, if called to do so, I am prepared and competent to testify on the matters presented in this declaration.

I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct. Executed on August 21, 2008, in Texarkana, Texas.



Dr. David Finkel

AUG-21-2008 23:04 P.17

TOTAL P.17

TABLE OF CONTENTS

Exh.	*Document*
1	Claim Chart U.S. Patent No. 5,894,554 – The Accused Oracle Application Server Products
2	Claim Chart U.S. Patent No. 6,415,335 – The Accused Oracle Application Server Products
3	Claim Chart U.S. Patent No. 5,894,554 – The Accused Oracle Database Products
4	Claim Chart U.S. Patent No. 6,415,335 – The Accused Oracle Database Products
5	Claim Chart U.S. Patent 5,894,554 – Accused Oracle Web Cache Products
6	Claim Chart U.S. Patent No. 6,415,335 – The Accused Oracle Web Cache Products
7	Oracle WebServer User's Guide Release 1.0 Part Number A349862" (153)
8	Popp et al. U.S. Patent No. 6,249,291 (195)
9	Oracle WebServer User's Guide Release 2.0.2 Production, Part No. A23646-2 (604/605)
10	Oracle WebServer User's Guide Release 2.0 Production, Part No. A23646-1 and Oracle WebServer 2.0 Technical Note
11	Functional Specification for Runtime Load Balancing in OCI Session Pools, 11gR1, 12/2/05, CHID00081-00105
12	Excerpt from deposition transcript of Lakshminarayanan Chidambaran, Feb. 29, 2008
13	Excerpt from deposition transcript of Carol Colrain, April 15, 2008
14	Excerpt from deposition transcript of Hal Hildebrand, Dec. 21, 2007
15	Excerpt from deposition transcript of Matthew Mayerson, April 15, 2008
16	Excerpt from deposition transcript of Mark Nelson, Feb. 22, 2008
17	Excerpt from deposition transcript of Douglas Surber, Feb. 28, 2008
18	Design Specification for OCI Runtime Connection Load Balancing in 11g, 12/14/05, ORCL01310468-01310498
19	Functional Specification for OFCGI, OHS, iAS, 11AS, 6/2/06, ORCL01209371-01209387

20 Functional Specification for Runtime Load Balancing in OCI Session Pools, 11gR1, 12/2/05, ORCL01166752-01166776

21 Functional Specification for Runtime Load Balancing, RDBMS, 10gR2, 4/1/04, ORCL01060148-01060193

22 Functional/Design Specification for Runtime Work Request Load Balancing using RAC Service Metrics, JDBC, 10gR2, 5/1/04, ORCL01054844-01054858

23 Excerpt from Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide 10g (9.0.4), B10320-01 Sept. 2003, ORCL00337734-00338101

24 Excerpt from Oracle Application Server Containers for J2EE, Support for JavaServer Pages Developer's Guide 10g Release 2 (10.1.2) B14014-02, July 2005, ORCL00444343-00444558

25 Excerpt from Oracle Application Server Web Cache Administrator's Guide 10g Release 2 (10.1.2) B14045-01, Dec. 2004, ORCL00473184-00473739

26 Excerpt from Oracle Call Interface Programmer's Guide, 11g Release 1 (11.1), B28395-02, Oct. 2007

27 Excerpt from Oracle Call Interface Programmer's Guide, 11g Release 1 (11.1), B28395-03, May 2008

28 Excerpt from Oracle Database Administrator's Guide, 10g Release 2 (10.2), B14231-02, May 2006, ORCL01969895-01970763

29 Excerpt from Oracle HTTP Server Administrator's Guide 10g (9.0.4), B10381-02, March 2004, ORCL00372061-00372362

30 Oracle Real Application Clusters 11g: An Oracle Technical White Paper, April 2007, ORCL01972320-01972333

31 Excerpt from Oracle Web Cache Administration and Deployment Guide, Release 1.0.2, A86722-01, Nov. 21, 2000, ORCL00107955-00108126

32 Excerpt from Oracle XML DB Developer's Guide 11g Release 1 (11.1), October 2007, ORCL01972631-01973585

33 Excerpt from Oracle9i Application Serer: Availability, Scalability and Manageability of J2EE and Web Cache Clusters, An Oracle White Paper, May 2002 ORCL00013356-00013398

34 Excerpt from Oracle9i Application Server Performance Guide, Release 2 (9.0.2), April 2002, A95102-02, ORCL00847081-00847294

35 Excerpt from Oracle9i Application Server, An Oracle White Paper, Sept. 2002, ORCL00010730-00010792

36 Excerpt from Oracle9i Application Server, Best Practices, Release 1 (v1.0.2.2) Sept. 2001 A95201-01, ORCL00190234-00190397

37 Oracle9i Application Server, ORCL00110683

38 Excerpt from Oracle9i JDBC Developer's Guide and Reference Release 2 (9.2) March 2002, ORCL00306997-00307580

39 Excerpt from Oracle9iAS Web Cache Administration and Deployment Guide, Release 2 (9.0.2), May 2002, Part No. A95404-02, ORCL00272342-00272887

40 Excerpt from Oracle9iAX Web Cache Administration and Deployment Guide, Release 2.0.0, Oct. 10, 2001, A90372-04, ORCL00189864-00190165

41 Presentation, Mod_oc4j Load Balancing in iAS 9.0.4, ORCL00018115-00018150

42 Excerpt from Plaintiff's Responses to Defendants' First Set of Interrogatories (1-6)

43 Tanenbaum, A.S., *Computer Networks*, Fourth Edition. Prentice-Hall PTR, Upper Saddle River, NJ, 2003.

44 Oracle WebServer User's Guide Release 1.0, A349862 ORCL00941713-00941960 (MAC003037-003284) (*Ref 153*)

45 US 6,651,108, Popp et al., MAC003907-003927 (*Ref. 194*)

46 US 6,249,291, Popp et al., MAC003928-003968 (*Ref. 195*)

47 Oracle WebServer User's Guide Release 2.0.2 Production, Part No. A23646-2, 1996, ORCL00000782-00001035 (*Ref. 604*)

48 Oracle WebServer 2.0 Technical Note, March 1996, ORCL00000761-00000781 (*Ref. 605*)

49 Oracle and the Internet, Using Oracle in the Emerging Electronic Infrastructure of Business and Society, July 1995, ORCL00937779-00937806

50 Excerpt from Understanding SQL*Net, Release 2.3, A42484-1, 1996, ORCL01808597-01808820

51 The Rush to the Net, May 1996, ORCL01606514-01606526

52 Web Request Broker API, Ver. 0.8, Feb. 12, 1996, ORCL01606852-01606867

53 Excerpt from deposition transcript of Katrina Montinola, Feb. 19, 2008

IN THE UNITED STATES DISTRICT COURT
FOR THE DISTRICT OF DELAWARE

CERTIFICATE OF SERVICE

I, David E. Moore, hereby certify that on August 29, 2008, the attached document was electronically filed with the Clerk of the Court using CM/ECF which will send notification to the registered attorney(s) of record that the document has been filed and is available for viewing and downloading.

I further certify that on August 29, 2008, the attached document was Electronically Mailed to the following person(s):

Mary B. Graham
Morris, Nichols, Arsht & Tunnell LLP
1201 North Market Street
P.O. Box 1347
Wilmington, DE 19899
MGraham@MNAT.com

James G. Gilliland
Igor Shoiket
Townsend and Townsend and Crew LLP
Two Embarcadero Center
Eighth Floor
San Francisco, CA 94111-3834
OracleEpicrealm@townsend.com

Theodore T. Herhold
Robert J. Artuz
Eric M. Hutchins
Eric A. Mercer
Joseph A. Greco
Nitin Gupta
Townsend and Townsend and Crew LLP
379 Lytton Avenue
Palo Alto, CA 94301
OracleEpicrealm@townsend.com

Chad E. King
Townsend and Townsend and Crew LLP
1200 Seventeenth Street
Suite 2700
Denver, CO 80202
OracleEpicrealm@townsend.com

/s/ David E. Moore
Richard L. Horwitz
David E. Moore
Potter Anderson & Corroon LLP
Hercules Plaza – Sixth Floor
1313 North Market Street
P.O. Box 951
Wilmington, DE 19899-0951
(302) 984-6000
rhorwitz@potteranderson.com
dmoore@potteranderson.com

Exhibit 11

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 12

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 13

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 14

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 15

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 16

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 17

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 18

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 19

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 20

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 21

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 22

THIS EXHIBIT HAS BEEN REDACTED IN ITS ENTIRETY

Exhibit 23

Oracle® Application Server Containers for J2EE

Support for JavaServer Pages Developer's Guide

10*g* (9.0.4)

Part No. B10320-01

September 2003

ORACLE

Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide, 10*g* (9.0.4)

Part No. B10320-01

Copyright © 2000, 2003 Oracle Corporation. All rights reserved.

Primary Author: Brian Wright

Contributing Author: Michael Freedman

Contributors: Julie Basu, Alex Yiu, Sunil Kunisetty, Gael Stevens, Sumathi Gopalakrishnan, Ping Guo, Olga Peschansky, John O'Duinn, YaQing Wang, Song Lin, Helen Zhao, Hal Hildebrand, Jasen Minton, Ashok Banerjee, Matthieu Devin, Jose Alberto Fernandez, Jerry Schwarz, Clement Lai, Shinji Yoshida, Kenneth Tang, Robert Pang, Kannan Muthukkaruppan, Ralph Gordon, Shiva Prasad, Sharon Malek, Jeremy Lizt, Kuassi Mensah, Susan Kraft, Sheryl Maring, Ellen Barnes, Angie Long, Sanjay Singh

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent and other intellectual and industrial property laws. Reverse engineering, disassembly or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Oracle9*i*, PL/SQL, and Oracle Store are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

4

Basic Programming Considerations

This chapter discusses basic programming considerations for JSP pages, including JSP-servlet interaction and database access, with examples provided.

The following sections are included:

- JSP-Servlet Interaction
- JSP Data-Access Support and Features
- JSP Resource Management
- Runtime Error Processing

JSP-Servlet Interaction

Although coding JSP pages is convenient in many ways, some situations call for servlets. One example is when you are outputting binary data, as discussed in "Reasons to Avoid Binary Data in JSP Pages" on page 6-16.

Therefore, it is sometimes necessary to go back and forth between servlets and JSP pages in an application. The following sections discuss how to accomplish this:

- Invoking a Servlet from a JSP Page
- Passing Data to a Servlet Invoked from a JSP Page
- Invoking a JSP Page from a Servlet
- Passing Data Between a JSP Page and a Servlet
- JSP-Servlet Interaction Samples

> **Important:** This discussion assumes a servlet 2.2 or higher environment, such as OC4J (servlet 2.3). Appropriate reference is made to other sections of this document for related considerations for Apache JServ and other servlet 2.0 environments.

Invoking a Servlet from a JSP Page

As when invoking one JSP page from another, you can invoke a servlet from a JSP page through the `jsp:include` and `jsp:forward` action tags. (See "Standard Actions: JSP Tags" on page 1-16.) Following is an example:

```
<jsp:include page="/servlet/MyServlet" flush="true" />
```

When this statement is encountered during page execution, the page buffer is output to the browser and the servlet is executed. When the servlet has finished executing, control is transferred back to the JSP page and the page continues executing. This is the same functionality as for `jsp:include` actions from one JSP page to another.

And as with `jsp:forward` actions from one JSP page to another, the following statement would clear the page buffer, terminate the execution of the JSP page, and execute the servlet:

```
<jsp:forward page="/servlet/MyServlet" />
```

Important: You cannot include or forward to a servlet in JServ or other servlet 2.0 environments; you would have to write a JSP wrapper page instead. For information, see "Dynamic Includes and Forwards in JServ" on page B-19.

Passing Data to a Servlet Invoked from a JSP Page

When dynamically including or forwarding to a servlet from a JSP page, you can use a `jsp:param` tag to pass data to the servlet (the same as when including or forwarding to another JSP page).

You can use a `jsp:param` tag within a `jsp:include` or `jsp:forward` tag. Consider the following example:

```
<jsp:include page="/servlet/MyServlet" flush="true" >
   <jsp:param name="username" value="Smith" />
   <jsp:param name="userempno" value="9876" />
</jsp:include>
```

For more information about the `jsp:param` tag, see "Standard Actions: JSP Tags" on page 1-16.

Alternatively, you can pass data between a JSP page and a servlet through a JavaBean of appropriate scope or through attributes of the HTTP request object. Using attributes of the request object is discussed later, in "Passing Data Between a JSP Page and a Servlet" on page 4-4.

Invoking a JSP Page from a Servlet

You can invoke a JSP page from a servlet through functionality of the standard `javax.servlet.RequestDispatcher` interface. Complete the following steps in your code to use this mechanism:

1. Get a servlet context instance from the servlet instance:

   ```
   ServletContext sc = this.getServletContext();
   ```

2. Get a request dispatcher from the servlet context instance, specifying the page-relative or application-relative path of the target JSP page as input to the `getRequestDispatcher()` method:

   ```
   RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
   ```

Prior to or during this step, you can optionally make data available to the JSP page through attributes of the HTTP request object. See "Passing Data Between a JSP Page and a Servlet" below for information.

3. Invoke the `include()` or `forward()` method of the request dispatcher, specifying the HTTP request and response objects as arguments. For example:

```
rd.include(request, response);
```

or:

```
rd.forward(request, response);
```

The functionality of these methods is similar to that of `jsp:include` and `jsp:forward` tags. The `include()` method only temporarily transfers control; execution returns to the invoking servlet afterward.

Note that the `forward()` method clears the output buffer.

Note: The request and response objects would have been obtained earlier, using standard servlet functionality such as the `doGet()` method specified in the `javax.servlet.http.HttpServlet` class.

Passing Data Between a JSP Page and a Servlet

The preceding section, "Invoking a JSP Page from a Servlet", notes that when you invoke a JSP page from a servlet through the request dispatcher, you can optionally pass data through the HTTP request object. You can accomplish this using either of the following approaches:

- You can append a query string to the URL when you obtain the request dispatcher, using "?" syntax with *`name=value`* pairs. For example:

```
RequestDispatcher rd =
      sc.getRequestDispatcher("/jsp/mypage.jsp?username=Smith");
```

In the target JSP page (or servlet), you can use the `getParameter()` method of the implicit `request` object to obtain the value of a parameter set in this way.

- You can use the `setAttribute()` method of the HTTP request object. For example:

```
request.setAttribute("username", "Smith");
RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
```

  In the target JSP page or servlet, you can use the `getAttribute()` method of the implicit `request` object to obtain the value of a parameter set in this way.

> **Note:** You can use the mechanisms discussed in this section instead of the `jsp:param` tag to pass data from a JSP page to a servlet.

JSP-Servlet Interaction Samples

This section provides a JSP page and a servlet that use functionality described in the preceding sections. The JSP page `Jsp2Servlet.jsp` includes the servlet `MyServlet`, which includes another JSP page, `welcome.jsp`.

Code for Jsp2Servlet.jsp

```
<HTML>
<HEAD> <TITLE> JSP Calling Servlet Demo </TITLE> </HEAD>
<BODY>

<!-- Forward processing to a servlet -->
<% request.setAttribute("empid", "1234"); %>
<jsp:include page="/servlet/MyServlet?user=Smith" flush="true"/>

</BODY>
</HTML>
```

Code for MyServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;
```


ORCL00337844

```
public class MyServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
      throws IOException, ServletException {
      PrintWriter out= response.getWriter();
      out.println("<B><BR>User:" + request.getParameter("user"));
      out.println
          (", Employee number:" + request.getAttribute("empid") + "</B>");
      this.getServletContext().getRequestDispatcher
                       ("/jsp/welcome.jsp").include(request, response);
    }
}
```

Code for welcome.jsp

```
<HTML>
<HEAD> <TITLE> The Welcome JSP  </TITLE> </HEAD>
<BODY>

<H3> Welcome! </H3>
<P><B> Today is <%= new java.util.Date() %>.  Have a nice day! </B></P>
</BODY>
</HTML>
```

JSP Data-Access Support and Features

The following sections discuss OC4J JSP and Oracle features to consider when accessing data:

- Introduction to JSP Support for Data Access
- JSP Data-Access Sample Using JDBC
- Use of JDBC Performance Enhancement Features
- EJB Calls from JSP Pages
- JSP Support for Oracle SQLJ
- OracleXMLQuery Class

Introduction to JSP Support for Data Access

Because the JDBC API is simply a set of Java interfaces, JavaServer Pages technology directly supports its use within JSP scriptlets.

Oracle JDBC provides several driver alternatives: 1) the JDBC OCI driver for use with an Oracle client installation; 2) a 100%-Java JDBC Thin driver that can be used in essentially any client situation, including applets; 3) a JDBC server-side Thin driver to access one Oracle Database instance from within another Oracle Database instance; and 4) a JDBC server-side internal driver to access the database within which the Java code is running, such as from a Java stored procedure. It is assumed that you are already at least somewhat familiar with JDBC basics, but you can refer to the *Oracle9i JDBC Developer's Guide and Reference*.

The OC4J JSP container also supports EJB calls as well as SQLJ (embedded SQL in Java).

Additionally, there are SQL tags in the JavaServer Pages Standard Tag Library (JSTL), and JavaBeans and custom SQL tags supplied with OC4J. These are all documented in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

JSP Data-Access Sample Using JDBC

The following example creates a query dynamically from search conditions the user enters through an HTML form (typed into a box, and entered with an `Ask Oracle` button). To perform the specified query, it uses JDBC code in a method called `runQuery()` that is defined in a JSP declaration. It also defines a method,

`formatResult()`, within the JSP declaration to produce the output. The `runQuery()` method uses the `scott` schema with password `tiger`.

The HTML `INPUT` tag specifies that the string entered in the form be named `cond`. Therefore, `cond` is also the input parameter to the `getParameter()` method of the implicit `request` object for this HTTP request, and the input parameter to the `runQuery()` method (which puts the `cond` string into the `WHERE` clause of the query).

Notes:

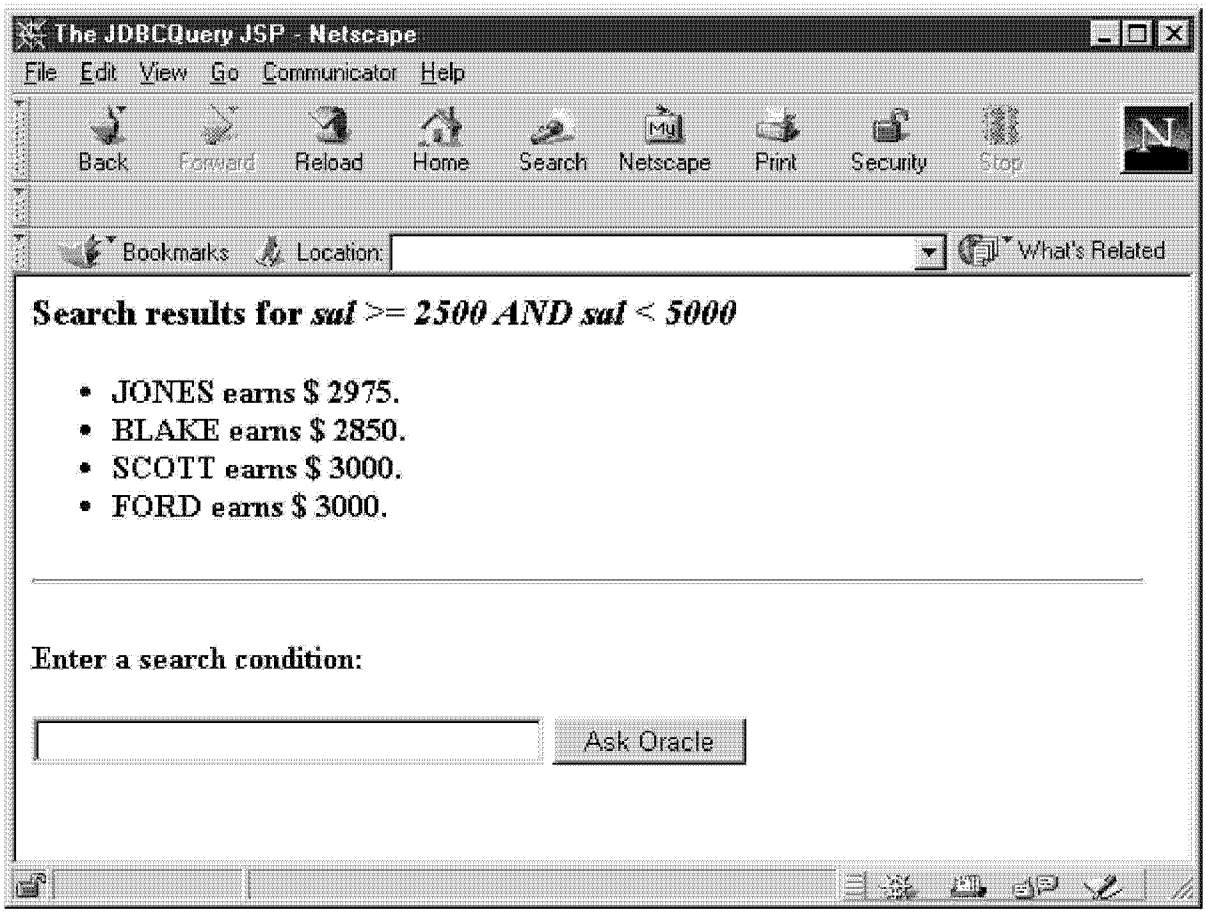
- Another approach to this example would be to define the `runQuery()` method in `<%...%>` scriptlet syntax instead of `<%!...%>` declaration syntax.
- This example uses the JDBC OCI driver, which requires an Oracle client installation. If you want to run this sample, use an appropriate JDBC driver and connection string.

```
<%@ page language="java" import="java.sql.*" %>
<HTML>
<HEAD> <TITLE> The JDBCQuery JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String searchCondition = request.getParameter("cond");
   if (searchCondition != null) { %>
      <H3> Search results for  <I> <%= searchCondition %> </I> </H3>
      <B> <%= runQuery(searchCondition) %> </B> <HR><BR>
<% }  %>
<B>Enter a search condition:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="cond" SIZE=30>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%-- Declare and define the runQuery() method. --%>
<%! private String runQuery(String cond) throws SQLException {
     Connection conn = null;
     Statement stmt = null;
     ResultSet rset = null;
     try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection("jdbc:oracle:oci:@",
                                           "scott", "tiger");
```

```
        stmt = conn.createStatement();
        // dynamic query
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                          (cond.equals("") ? "" : "WHERE " + cond ));
      return (formatResult(rset));
    } catch (SQLException e) {
        return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    } finally {
        if (rset!= null) rset.close();
        if (stmt!= null) stmt.close();
        if (conn!= null) conn.close();
    }
  }
  private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
      sb.append("<P> No matching rows.<P>\n");
    else {  sb.append("<UL>");
            do {  sb.append("<LI>" + rset.getString(1) +
                            " earns $ " + rset.getInt(2) + ".</LI>\n");
            } while (rset.next());
           sb.append("</UL>");
    }
    return sb.toString();
  }
%>
```

The graphic below illustrates sample output for the following input:

```
sal >= 2500 AND sal < 5000
```



Use of JDBC Performance Enhancement Features

JSP applications in OC4J can use features for the following performance enhancements, supported through Oracle JDBC extensions:

- Caching database connections
- Caching JDBC statements
- Batching update statements

- Prefetching rows during a query
- Caching rowsets

Most of these performance features are supported by the Oracle `ConnBean` and `ConnCacheBean` data-access JavaBeans (but not by `DBBean`). These beans are described in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Database Connection Caching

Creating a new database connection is an expensive operation that you should avoid whenever possible. Instead, use a cache of database connections. A JSP application can get a logical connection from a pre-existing pool of physical connections, and return the connection to the pool when done.

You can create a connection pool at any one of the four JSP scopes—`application`, `session`, `page`, or `request`. It is most efficient to use the maximum possible scope—`application` scope if that is permitted by the Web server, or `session` scope if not.

The Oracle JDBC connection caching scheme, built upon standard connection pooling as specified in the JDBC 2.0 standard extensions, is implemented in the `ConnCacheBean` data-access JavaBean provided with OC4J. Alternatively, you can use standard data-source connection pooling functionality, which is supported by the `ConnBean` data-access JavaBean. These beans are described in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

It is also possible to use the Oracle JDBC `OracleConnectionCacheImpl` class directly, as though it were a JavaBean, as in the following example (although all `OracleConnectionCacheImpl` functionality is available through `ConnCacheBean`):

```
<jsp:useBean id="occi" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
             scope="session" />
```

The same properties are available in `OracleConnectionCacheImpl` as in `ConnCacheBean`. They can be set either through `jsp:setProperty` tags or directly through the class setter methods.

For information about the Oracle JDBC connection caching scheme and the `OracleConnectionCacheImpl` class, see the *Oracle9i JDBC Developer's Guide and Reference*.

JDBC Statement Caching

Statement caching, an Oracle JDBC extension, improves performance by caching executable statements that are used repeatedly within a single physical connection, such as in a loop or in a method that is called repeatedly. When a statement is cached, the statement does not have to be re-parsed, the statement object does not have to be re-created, and parameter size definitions do not have to be recalculated each time the statement is executed.

The Oracle JDBC statement caching scheme is implemented in the `ConnBean` and `ConnCacheBean` data-access JavaBeans that are provided with OC4J. Each of these beans has a `stmtCacheSize` property that can be set through a `jsp:setProperty` tag or the bean `setStmtCacheSize()` method. The beans are described in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Statement caching is also available directly through the Oracle JDBC `OracleConnection` and `OracleConnectionCacheImpl` classes. For information about the Oracle JDBC statement caching scheme and the `OracleConnection` and `OracleConnectionCacheImpl` classes, see the *Oracle9i JDBC Developer's Guide and Reference*.

> **Important:** Statements can be cached only within a single physical connection. When you enable statement caching for a connection cache, statements can be cached across multiple logical connection objects from a single pooled connection object, but not across multiple pooled connection objects.

Update Batching

The Oracle JDBC update batching feature associates a batch value (limit) with each prepared statement object. With update batching, instead of the JDBC driver executing a prepared statement each time its execution method is called, the driver adds the statement to a batch of accumulated execution requests. The driver will pass all the operations to the database for execution once the batch value is reached. For example, if the batch value is 10, then each batch of ten operations will be sent to the database and processed in one trip.

OC4J supports Oracle JDBC update batching directly, through the `executeBatch` property of the `ConnBean` data-access JavaBean. You can set this property through a `jsp:setProperty` tag or through the setter method of the bean. If you use `ConnCacheBean` instead, you can enable update batching through Oracle JDBC functionality in the connection and statement objects you create. These beans are

described in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For more information about Oracle JDBC update batching, see the *Oracle9i JDBC Developer's Guide and Reference*.

Row Prefetching

For the population of query result sets, the Oracle JDBC row prefetching feature enables you to determine the number of rows to prefetch into the client during each trip to the database. This reduces the number of round-trips to the server.

OC4J supports Oracle JDBC row prefetching directly, through the `preFetch` property of the `ConnBean` data-access JavaBean. You can set this property through a `jsp:setProperty` tag or through the setter method of the bean. If you use `ConnCacheBean` instead, you can enable row prefetching through Oracle JDBC functionality in the connection and statement objects you create. These beans are described in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For more information about Oracle JDBC row prefetching, see the *Oracle9i JDBC Developer's Guide and Reference*.

Rowset Caching

A cached rowset provides a disconnected, serializable, and scrollable container for retrieved data. This feature is useful for small sets of data that do not change often, particularly when the client requires frequent or continued access to the information. By contrast, using a normal result set requires the underlying connection and other resources to be held. Be aware, however, that large cached rowsets consume a lot of memory on the application server.

In Oracle Database, the Oracle JDBC implementation provides a cached rowset implementation. If you are using an Oracle JDBC driver, use code inside a JSP page to create and populate a cached rowset, as follows:

```
CachedRowSet crs = new CachedRowSet();
crs.populate(rset); // rset is a previously created JDBC ResultSet object.
```

Once the rowset is populated, the connection and statement objects used in obtaining the original result set can be closed.

For more information about Oracle JDBC cached rowsets, see the *Oracle9i JDBC Developer's Guide and Reference*.

EJB Calls from JSP Pages

JSP pages can call EJBs to perform additional processing or data access. A typical application design uses JavaServer Pages as a front-end for the initial processing of client requests, with Enterprise JavaBeans being called to perform the work that involves reading from and writing to data sources. The following sections provide an overview of EJB usage:

- Overview of Configuration and Deployment for EJBs
- Code Steps and Approaches for EJB Calls
- Use of the OC4J EJB Tag Library

Overview of Configuration and Deployment for EJBs

The configuration and deployment steps for calling EJBs from JSP pages are similar to the steps for calling EJBs from servlets, which are described in the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*. These steps include the following:

- Define an `<ejb-ref>` element in the application `web.xml` file for each EJB called from a JSP page.
- Create an `ejb-jar.xml` deployment descriptor that contains an `<enterprise-beans>` element with appropriate subelements, such as `<session>` or `<entity>`, that specify the types of EJBs. Within these subelements, specify the name, class name, and other details for each called EJB.
- Package the `ejb-jar.xml` file in the EJB archive. Deployment requirements are very similar to the requirements for servlets.

Code Steps and Approaches for EJB Calls

The key steps required for a JSP page to invoke an EJB are the following:

1. Import the EJB package for the bean home and remote interfaces into each JSP page that makes EJB calls. Use a `page` directive for this.
2. Use JNDI to look up the EJB home interface.
3. Create the EJB remote object from the home.
4. Invoke business methods on the remote object.

Because you can use almost any servlet code in a JSP page in the form of a scriptlet, one straightforward way to call EJBs from a JSP page is to use the same code in a

scriptlet that you would use in a servlet. This is one way to accomplish steps 2, 3, and 4.

Alternatively, you can use tags from the EJB tag library provided with OC4J. This is described in the next section, "Use of the OC4J EJB Tag Library". These tags simplify the coding. Essentially, they allow you to treat Enterprise JavaBeans similarly to regular JavaBeans, which are commonly used in JSP pages.

Use of the OC4J EJB Tag Library

Refer to the preceding section, "Code Steps and Approaches for EJB Calls". As in that section, import the appropriate package in a `page` directive. Then use the OC4J EJB tags as follows:

- Use a `taglib` directive to specify the tag prefix and the tag library descriptor (TLD) file that you will use.
- For step 2 of the code steps, use an EJB `useHome` tag.
- For step 3 of the code steps, you can use an EJB `createBean` tag inside an EJB `useBean` tag.
- For step 4 of the code steps, the EJB `iterate` tag enables you to apply business methods to each member of a collection of EJB objects, usually returned by a `find` method.

For more information about the EJB tag library, including detailed tag syntax, see the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Deployment requirements are the same for the tag library approach as for the scriptlet code approach. As with any tag library, the TLD and the library support classes (tag handler classes and tag-extra-info classes) must be made accessible to your application.

JSP Support for Oracle SQLJ

Important: SQLJ is being desupported after the current release. Consult MetaLink, at the following location, for the desupport notice:

```
http://metalink.oracle.com/
```

SQLJ is a standard syntax for embedding static SQL instructions directly in Java code, greatly simplifying database-access programming. The OC4J JSP container supports the Oracle SQLJ implementation, allowing you to use SQLJ syntax in JSP statements. SQLJ statements are indicated by the `#sql` token. Oracle SQLJ database access is typically through the Oracle JDBC drivers.

For general information about Oracle SQLJ programming features, syntax, and command-line options, see the *Oracle9i SQLJ Developer's Guide and Reference*.

SQLJ JSP Code Example

Following is a sample SQLJ JSP page. The page directive imports classes that are typically required by SQLJ.

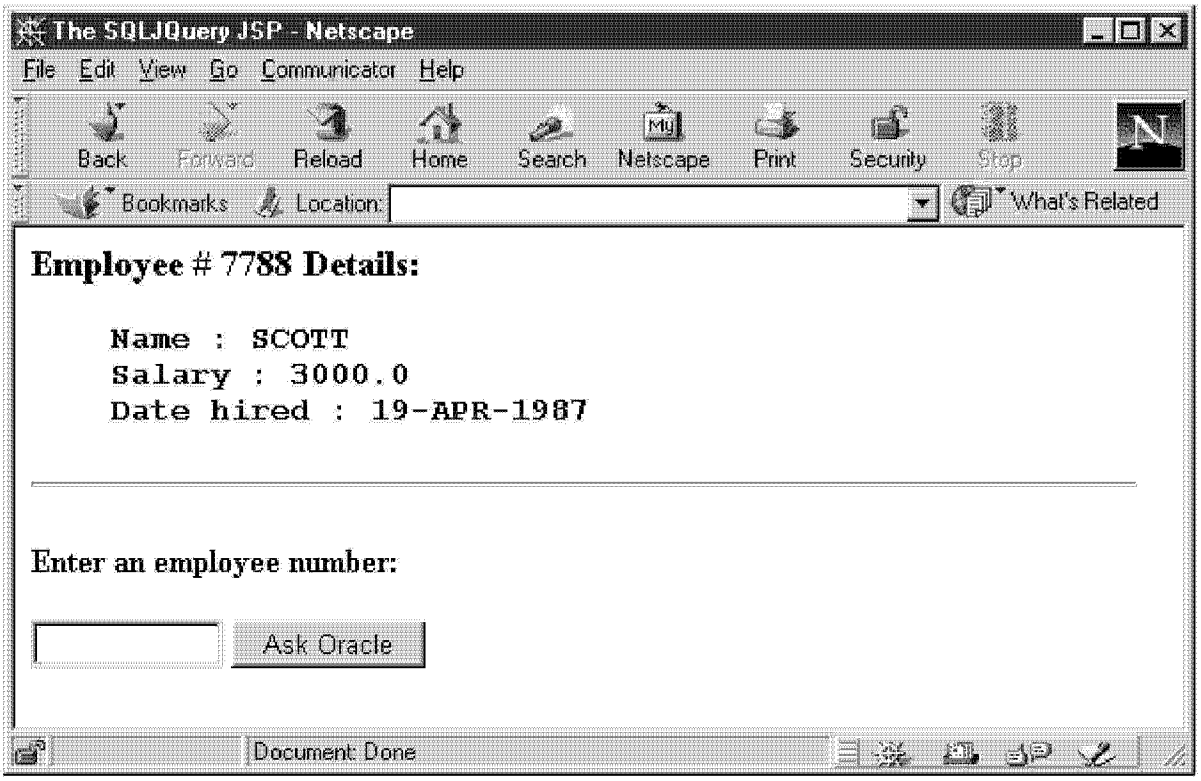
```
<%@ page language="sqlj"
    import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>
<HTML>
<HEAD> <TITLE> The SQLJQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String empno = request.getParameter("empno");
if (empno != null) { %>
<H3> Employee # <%=empno %> Details: </H3>
<%= runQuery(empno) %>
<HR><BR>
<% } %>
<B>Enter an employee number:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!
```

```
private String runQuery(String empno) throws java.sql.SQLException {
   DefaultContext dctx = null;
   String ename = null; double sal = 0.0; String hireDate = null;
   StringBuffer sb = new StringBuffer();
   try {
     dctx = Oracle.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");
     #sql [dctx] {
       SELECT ename, sal, TO_CHAR(hiredate,'DD-MON-YYYY')
       INTO :ename, :sal, :hireDate
       FROM scott.emp WHERE UPPER(empno) = UPPER(:empno) };
     sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
     sb.append("Name : " + ename + "\n");
     sb.append("Salary : " + sal + "\n");
     sb.append("Date hired : " + hireDate);
     sb.append("</PRE></B></BIG></BLOCKQUOTE>");
     } catch (java.sql.SQLException e) {
       sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
     } finally {
     if (dctx!= null) dctx.close();
     }
  return sb.toString();
}

%>
```

This example uses the JDBC OCI driver, which requires an Oracle client installation. The `Oracle` class used in getting the connection is provided with the Oracle SQLJ implementation.

Entering employee number 7788 results in output such as the following:



Notes:

- In case a JSP page is invoked multiple times in the same JVM, it is recommended that you always use an explicit connection context, such as `dctx` in the example, instead of the default connection context. (Note that `dctx` is a local method variable.)
- If you use the Oracle SQLJ implementation, the OC4J JSP container requires SQLJ release 8.1.6.1 or higher.

Triggering the SQLJ Translator

You can trigger the OC4J JSP translator to invoke the Oracle SQLJ translator in one of two ways:

- Use the file name extension `.sqljsp` for the JSP source file.

or:

- Specify `language="sqlj"` in a `page` directive.

Either of these results in the JSP translator generating a `.sqlj` file instead of a `.java` file. The Oracle SQLJ translator is then invoked to translate the `.sqlj` file into a `.java` file.

Using SQLJ results in additional output files—see "Generated Files and Locations" on page 7-6.

Important:

- To use SQLJ, you must install the SQLJ JAR files that are appropriate for your environment and add them to your classpath. See "Key Support Files Provided with OC4J" on page 3-12.
- Do not use the same base file name for a `.jsp` file and a `.sqljsp` file in the same application, because this would result in duplicate generated class names and `.java` file names.

Setting Oracle SQLJ Options

When you execute or pretranslate a SQLJ JSP page, you can specify desired Oracle SQLJ option settings. This is true both in on-demand translation scenarios and pretranslation scenarios, as follows:

- In an on-demand translation scenario, use the JSP `sqljcmd` configuration parameter. This parameter, in addition to allowing you to specify a particular SQLJ translator executable, enables you to set SQLJ command-line options.

  For information about `sqljcmd`, see "JSP Configuration Parameters" on page 3-14.

- In a pretranslation scenario with the `ojspc` pretranslation tool, use the `ojspc -s` option. This option enables you to set SQLJ command-line options.

  For information, see "Command-Line Syntax for ojspc" on page 7-20 and "Option Descriptions for ojspc" on page 7-21.

OracleXMLQuery Class

The `oracle.xml.sql.query.OracleXMLQuery` class is part of the Oracle Database XML-SQL utility for XML functionality in database queries. This class requires file `xsu12.jar` (or `xsu111.jar` for JDK 1.1.x), which is also required for XML functionality in some of the custom tags and JavaBeans provided with OC4J. This file is provided with Oracle Database and Oracle Application Server.

For information about the `OracleXMLQuery` class and other XML-SQL utility features, refer to the *Oracle XML Developer's Kit Programmer's Guide*.

ORCL00337859

JSP Resource Management

The following sections discuss standard features and Oracle value-added features for resource management:

- Standard Session Resource Management: HttpSessionBindingListener
- Overview of Oracle Value-Added Features for Resource Management

Standard Session Resource Management: HttpSessionBindingListener

A JSP page must appropriately manage resources acquired during its execution, such as JDBC connection, statement, and result set objects. The standard `javax.servlet.http` package provides the `HttpSessionBindingListener` interface and `HttpSessionBindingEvent` class to manage session-scope resources. Through this mechanism, a session-scope query bean could, for example, acquire a database cursor when the bean is instantiated and close it when the HTTP session is terminated. (The example in "JSP Data-Access Sample Using JDBC" on page 4-7 opens and closes the connection for each query, which adds overhead.)

This section describes use of the `HttpSessionBindingListener` `valueBound()` and `valueUnbound()` methods.

> **Note:** The bean instance must register itself in the event notification list of the HTTP session object, but the `jsp:useBean` statement takes care of this automatically.

The valueBound() and valueUnbound() Methods

An object that implements the `HttpSessionBindingListener` interface can implement a `valueBound()` method and a `valueUnbound()` method, each of which takes an `HttpSessionBindingEvent` instance as input. These methods are called by the servlet container—the `valueBound()` method when the object is stored in the session, and the `valueUnbound()` method when the object is removed from the session or when the session reaches a timeout or becomes invalid. Usually, a developer will use `valueUnbound()` to release resources held by the object (in the example below, to release the database connection).

"JDBCQueryBean JavaBean Code" below provides a sample JavaBean that implements `HttpSessionBindingListener` and a sample JSP page that calls the bean.

JDBCQueryBean JavaBean Code

Following is the sample code for `JDBCQueryBean`, a JavaBean that implements the `HttpSessionBindingListener` interface. It uses the JDBC OCI driver for its database connection; use an appropriate JDBC driver and connection string if you want to run this example yourself.

`JDBCQueryBean` gets a search condition through the HTML request (as described in "UseJDBCQueryBean JSP Page" on page 4-24), executes a dynamic query based on the search condition, and outputs the result.

This class also implements a `valueUnbound()` method, as specified in the `HttpSessionBindingListener` interface, that results in the database connection being closed at the end of the session.

```
package mybeans;

import java.sql.*;
import javax.servlet.http.*;

public class JDBCQueryBean implements HttpSessionBindingListener
{
  String searchCond = "";
  String result = null;

  public void JDBCQueryBean() {
  }

  public synchronized String getResult() {
    if (result != null) return result;
    else return runQuery();
  }

  public synchronized void setSearchCond(String cond) {
    result = null;
    this.searchCond = cond;
  }

  private Connection conn = null;

  private String runQuery() {
    StringBuffer sb = new StringBuffer();
    Statement stmt = null;
    ResultSet rset = null;
```

```
    try {
      if (conn == null) {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                           "scott", "tiger");

      }

      stmt = conn.createStatement();
      rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
            (searchCond.equals("") ? "" : "WHERE " + searchCond ));
      result = formatResult(rset);
      return result;

    } catch (SQLException e) {
      return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    }
    finally {
      try {
        if (rset != null) rset.close();
        if (stmt != null) stmt.close();
      }
      catch (SQLException ignored) {}
    }
  }

  private String formatResult(ResultSet rset) throws SQLException  {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
      sb.append("<P> No matching rows.<P>\n");
    else {
      sb.append("<UL><B>");
      do {  sb.append("<LI>" + rset.getString(1) +
            " earns $ " + rset.getInt(2) + "</LI>\n");
      } while (rset.next());
      sb.append("</B></UL>");
    }
    return sb.toString();
  }

  public void valueBound(HttpSessionBindingEvent event) {
    // do nothing -- the session-scope bean is already bound
  }
```

```
  public synchronized void valueUnbound(HttpSessionBindingEvent event) {
    try {
      if (conn != null) conn.close();
    }
    catch (SQLException ignored) {}
  }
}
```

> **Note:** The preceding code serves as a sample only. This is not necessarily an advisable way to handle database connection pooling in a large-scale Web application.

UseJDBCQueryBean JSP Page

The following JSP page uses the `JDBCQueryBean` JavaBean defined in "JDBCQueryBean JavaBean Code" above, invoking the bean with `session` scope. It uses `JDBCQueryBean` to display employee names that match a search condition entered by the user.

`JDBCQueryBean` gets the search condition through the `jsp:setProperty` tag in this JSP page, which sets the `searchCond` property of the bean according to the value of the `searchCond` request parameter input by the user through the HTML form. The HTML `INPUT` tag specifies that the search condition entered in the form be named `searchCond`.

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="searchCond" />

<HTML>
<HEAD> <TITLE> The UseJDBCQueryBean JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("searchCond");
   if (searchCondition != null) { %>
      <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
      <%= queryBean.getResult() %>
      <HR><BR>
<% }  %>

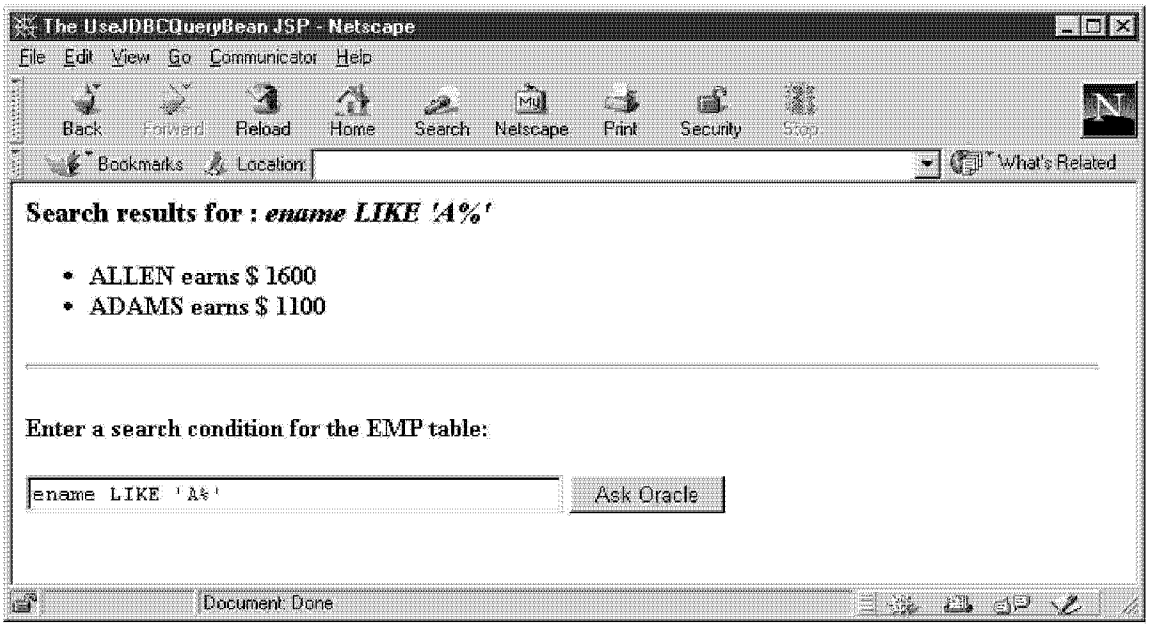
<B>Enter a search condition for the EMP table:</B>

<FORM METHOD="get">
<INPUT TYPE="text" NAME="searchCond" VALUE="ename LIKE 'A%' " SIZE="40">
```

```
<INPUT TYPE="submit" VALUE="Ask Oracle">
</FORM>

</BODY>
</HTML>
```

Following is sample input and output for this page:



Advantages of HttpSessionBindingListener

In the preceding example, an alternative to the `HttpSessionBindingListener` mechanism would be to close the connection in a `finalize()` method in the JavaBean. The `finalize()` method would be called when the bean is garbage-collected after the session is closed. The `HttpSessionBindingListener` interface, however, has more predictable behavior than a `finalize()` method. Garbage collection frequency depends on the memory consumption pattern of the application. By contrast, the `valueUnbound()` method of the `HttpSessionBindingListener` interface is called reliably at session shutdown.

ORCL00337864

Overview of Oracle Value-Added Features for Resource Management

OC4J JSP provides the `JspScopeListener` interface for managing application-scope, session-scope, request-scope, or page-scope resources in a servlet 2.3 environment such as OC4J.

This mechanism adheres to servlet and JSP standards in supporting objects of `page`, `request`, `session`, or `application` scope. To create a class that supports session scope as well as other scopes, you can integrate `JspScopeListener` with `HttpSessionBindingListener` by having the class implement both interfaces. For `page` scope in OC4J or JServ environments, you also have the option of using an Oracle-specific runtime implementation.

For information about configuration and how to integrate with `HttpSessionBindingListener`, see the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Runtime Error Processing

While a JSP page is executing and processing client requests, runtime errors can occur either inside the page or outside the page, such as in a called JavaBean. This section describes error processing mechanisms and provides an elementary example.

Servlet and JSP Runtime Error Mechanisms

This section describes servlet 2.3 and JSP 1.2 mechanisms for handling runtime exceptions, including the use of JSP error pages.

General Servlet Runtime Error Mechanism

Any runtime error encountered during execution of a JSP page is handled through the standard Java exception mechanism in one of two ways:

- You can catch and handle exceptions in a Java scriptlet within the JSP page itself, using standard Java exception-handling code.
- Exceptions that you do not catch in the JSP page will result in forwarding of the request and uncaught exception, a `java.lang.Throwable` instance, to an error resource. This is the preferred way to handle JSP errors. In this case, the exception instance describing the error is stored in the `request` object through a `setAttribute()` call, using `javax.servlet.jsp.jspException` as the name.

You can specify the URL of an error resource by setting the `errorPage` attribute in a `page` directive in the originating JSP page. (For an overview of JSP directives, including the `page` directive, see "Directives" on page 1-7.)

In a servlet 2.2 or higher environment, you can also specify a default error page in the `web.xml` deployment descriptor through instructions such as the following:

```
<error-page>
   <error-code>404</error-code>
   <location>/error404.html</location>
</error-page>
```

See the Sun Microsystems *Java Servlet Specification, Version 2.3* for more information about default error resources.

JSP Error Pages

You have the option of using another JSP page as the error resource for runtime exceptions from an originating JSP page. A JSP error page must have a `page` directive setting `isErrorPage="true"`. An error page defined in this way takes precedence over an error page declared in the `web.xml` file.

The `java.lang.Throwable` instance describing the error is accessible in the error page through the JSP implicit `exception` object. Only an error page can access this object. For information about JSP implicit objects, including the `exception` object, see "Implicit Objects" on page 1-13.

Be aware that if an originating JSP page has a `page` directive with `autoFlush="true"` (the default setting), and the contents of the `JspWriter` object from that page have already been flushed to the response output stream, then any further attempt to forward an uncaught exception to any error page might not be able to clear the response. Some of the response might have already been received by the browser.

See "JSP Error Page Example" below for an example of error page usage.

JSP Error Page Example

The following example, `nullpointer.jsp`, generates an error and uses an error page, `myerror.jsp`, to output contents of the implicit `exception` object.

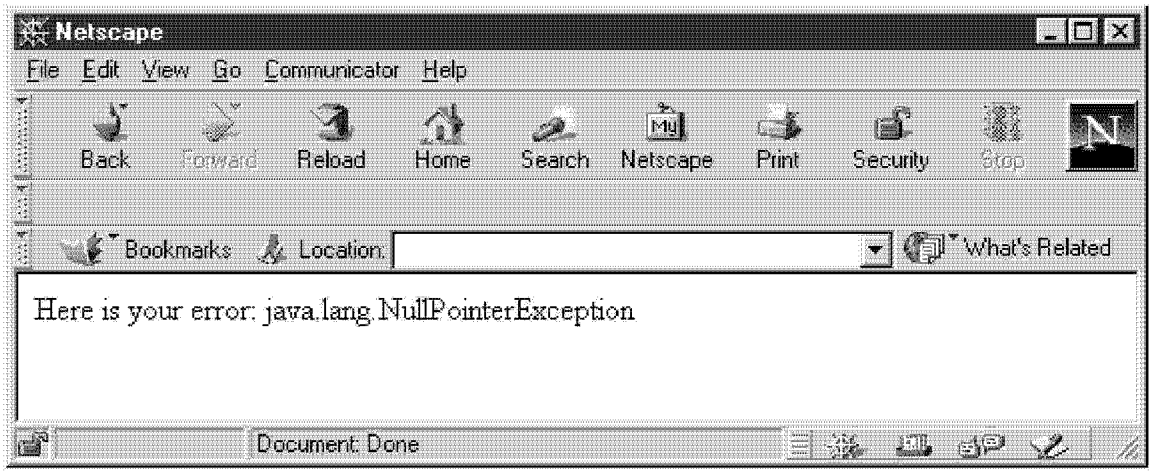
Code for nullpointer.jsp

```
<HTML>
<BODY>
<%@ page errorPage="myerror.jsp" %>
Null pointer is generated below:
<%
   String s=null;
   s.length();
%>
</BODY>
</HTML>
```

Code for myerror.jsp

```
<HTML>
<BODY>
<%@ page isErrorPage="true" %>
Here is your error:
<%= exception %>
</BODY>
</HTML>
```

This example results in the following output:



Note: The line "Null pointer is generated below:" in `nullpointer.jsp` is not output when processing is forwarded to the error page. This shows the difference between `jsp:include` and `jsp:forward` functionality. With `jsp:forward`, the output from the "forward-to" page *replaces* the output from the "forward-from" page.

ORCL00337869

Exhibit 24

Oracle® Application Server Containers for J2EE

Support for JavaServer Pages Developer's Guide

10*g* Release 2 (10.1.2)

B14014-02

July 2005

ORACLE

Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide, 10g Release 2 (10.1.2)

B14014-02

Copyright © 2000, 2005, Oracle. All rights reserved.

Primary Author: Dan Hynes, Brian Wright

Contributing Author: Michael Freedman

Contributors: Ashok Banerjee, Ellen Barnes, Julie Basu, Matthieu Devin, Jose Alberto Fernandez, Sumathi Gopalakrishnan, Ralph Gordon, Ping Guo, Hal Hildebrand, Susan Kraft, Sunil Kunisetty, Clement Lai, Song Lin, Jeremy Lizt, Angie Long, Sharon Malek, Sheryl Maring, Kuassi Mensah, Jasen Minton, Kannan Muthukkaruppan, John O'Duinn, Robert Pang, Olga Peschansky, Shiva Prasad, Jerry Schwarz, Sanjay Singh, Gael Stevens, Kenneth Tang, YaQing Wang, Alex Yiu, Shinji Yoshida, Helen Zhao

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Retek are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

2

Overview of the Oracle JSP Implementation

The JSP container provided with Oracle Application Server Containers for J2EE (OC4J) in the Oracle Application Server is a complete implementation of the JSP 1.2 specification. This functionality depends upon servlet 2.3 functionality, and the OC4J servlet container is a complete implementation of the servlet 2.3 specification.

This chapter provides overviews of the Oracle Application Server, OC4J, the OC4J JSP implementation and features, and custom tag libraries and utilities that are also supplied (documented in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*).

The following sections are included:

- Overview of the Oracle Application Server and JSP Support
- Oracle JDeveloper JSP Support
- Overview of Oracle Value-Added Features

Overview of the Oracle Application Server and JSP Support

The following sections provide a brief overview of the Oracle Application Server, its J2EE environment, its JSP implementation, and its Web server:

- Overview of the Oracle Application Server
- Oracle HTTP Server and mod_oc4j
- Overview of OC4J
- Overview of the JSP Implementation in OC4J

> **Note:** Users of earlier Oracle Application Server releases can refer to *Oracle Application Server Upgrade and Compatibility Guide* for information about issues in migrating to the current release.

Overview of the Oracle Application Server

Oracle Application Server is a scalable, secure, middle-tier application server. It can be used to deliver Web content, host Web applications, connect to back-office applications, and make these services accessible to any client browser. Users can access information, perform business analysis, and run business applications on the Internet or corporate intranets or extranets. Major areas of functionality include business intelligence, e-business integration, J2EE Web services, performance and caching, portals, wireless services, and management and security. For performance, scalability, and dependability, there are also clustering and load-balancing features.

To deliver this range of content and services, the Oracle Application Server incorporates many components, including the Oracle HTTP Server, Oracle Application Server Web Cache, Oracle Application Server Web Services, Oracle Application Server Portal, Oracle Application Server Wireless, and business logic runtime environments that support Enterprise JavaBeans, stored procedures, and Oracle Application Development Framework (Oracle ADF) Business Components.

For its Java environment, Oracle Application Server provides the Oracle Application Server Containers for J2EE (OC4J), a J2EE 1.3-compliant set of containers and services. This includes the JSP container described in this manual, a servlet container, and an EJB container.

For administration, you can fully manage and configure Oracle Application Server and OC4J using the HTML-based Oracle Enterprise Manager 10*g*. This includes full support for managing clustering, configuration, and deployment.

Oracle HTTP Server and mod_oc4j

Oracle HTTP Server, powered by the Apache Web server, is included with Oracle Application Server as the HTTP entry point for Web applications, particularly in a production environment. By default, it is the front-end for all OC4J processes. Client requests go through Oracle HTTP Server first.

When the Oracle HTTP Server is used, dynamic content is delivered through various Apache *mod* components provided either by the Apache Software Foundation or by Oracle. Static content is typically delivered from the file system, which is more efficient in this case. An Apache mod is typically a module of C code, running in the Apache address space, that passes requests to a particular mod-specific processor. The mod software will have been written specifically for use with the particular processor.

Oracle Application Server supplies the `mod_oc4j` Apache mod, which is used for communication between the Oracle HTTP Server and OC4J. It routes requests from the Oracle HTTP Server to OC4J processes, and forwards responses from OC4J processes to Web clients.

Communication is through the Apache JServ protocol (AJP). AJP was chosen over HTTP because of a variety of AJP features allowing faster communication, including use of binary format and more efficient processing of message headers.

The following features are provided with `mod_oc4j`:

- Load balancing capabilities across many back-end OC4J clusters
- Stateless session routing of stateful servlets

  This is accomplished through enhanced use of cookies. Routing information is maintained in the cookie itself to ensure that stateful servlets are always routed to the same OC4J JVM.

> **Note:** Oracle HTTP Server is not relevant for an OC4J standalone environment (typically used only during development). See "OC4J Standalone" on page 2-5.

Overview of OC4J

The following sections provide an overview of features of OC4J, the J2EE component of the Oracle Application Server:

- OC4J General Features

- OC4J Services
- OC4J Containers
- OC4J Standalone

OC4J General Features

OC4J is a high-performance, J2EE-compliant set of containers and services providing a scalable and reliable server infrastructure. In Oracle Application Server 10*g* Release 2 (10.1.2), OC4J complies with the J2EE 1.3 specification.

For developer convenience, OC4J has been integrated with Oracle JDeveloper and other development tools, and can run in a standalone mode separate from Oracle Application Server during the development process.

Java applications built with any development tool can be deployed to OC4J, which supports standard EAR or WAR file deployment. You can debug applications deployed to OC4J through standard Java profiling and debugging facilities.

For security, OC4J supports Secure Socket Layer (SSL) and HTTPS functionality.

> **Note:** Each OC4J instance runs in a single Java virtual machine.

OC4J Services

OC4J supports the following Java and J2EE services:

- J2EE Connector Architecture (JCA): JCA defines a standard architecture for connecting J2EE platforms to heterogeneous enterprise information systems such as ERP systems, mainframe transaction processing, database systems, and legacy applications.
- Java Transaction API (JTA) and two-phase commits: JTA allows simultaneous updates to multiple resources in a single, coordinated transaction.
- Java Message Service (JMS) integration: This integration allows compatibility between the Oracle JMS implementation and those of other JMS providers.
- Java Naming and Directory Interface (JNDI): JNDI associates names with resources for lookup purposes.
- Java Authentication and Authorization Service (JAAS): The Oracle implementation of JAAS and the Java2 security model provides complete support for development and deployment of secure applications and for fine-grained authorization and access control.
- JDBC data sources: This is the standard mechanism for connecting to a database.

See the *Oracle Application Server Containers for J2EE Services Guide* for information.

OC4J Containers

The OC4J 10.1.2 implementation supplies the following J2EE containers:

- A JSP container complying with the Sun JSP 1.2 specification

  The JSP bundle also supplies tag libraries to implement Web services, caching capabilities, SQL access, file access, and other features. For further overview of the JSP container provided with OC4J, see "Overview of the JSP Implementation in OC4J" on page 2-5.

ORCL00444379

- A servlet container complying with the servlet 2.3 specification (with key features described immediately below)
- An EJB container complying with the EJB 2.0 specification (with key features described below)

OC4J containers have been instrumented to support the Dynamic Monitoring Service (DMS) to provide runtime performance data. You can view this data through Enterprise Manager.

> **Note:** Servlet 2.3 compliance is required in order to support JSP 1.2 compliance.

Key Servlet Container Features The OC4J servlet container supports the following key features:

- SSL and HTTPS: In Oracle Application Server, OC4J supports SSL (Secure Socket Layer) communication between Oracle HTTP Server and OC4J, using secure AJP. In addition, OC4J standalone supports SSL communication directly between a client and OC4J, using HTTPS.
- Integration with SSO and Oracle Internet Directory: This is through the Oracle JAAS implementation.
- Stateful failover and cluster deployment: For a distributable application, session state is replicated to alternate OC4J servers so that state is not lost in the event of failover.
- Servlet filtering: This allows transformation of the content of an HTTP request or response, and modification of header information.
- Application-level and session-level event listeners: This feature allows greater control over interaction with servlet context and HTTP session objects and, therefore, greater efficiency in managing resources that the application uses.

See the *Oracle Application Server Containers for J2EE Servlet Developer's Guide* for information.

Key EJB Container Features The OC4J EJB container supports the following:

- Session beans: A session bean is used for task-oriented requests. You can define a session bean as stateless or stateful. Stateless session beans cannot maintain state information across calls, while stateful session beans *can* maintain state across calls.
- Entity beans: An entity bean represents data. It can use the container to maintain the data persistently, which is referred to as *container-managed persistence* (CMP), or it can use the bean implementation to manage the data, which is referred to as *bean-managed persistence* (BMP).
- Message-driven beans (MDB): A message-driven bean is used to receive JMS messages from a queue or topic. It can then invoke other EJBs to process the JMS message.

EJB support in OC4J also includes these features:

- Clustering for session and entity beans
- Enhanced entity bean concurrency models to support concurrent access from multiple clients

ORCL00444380

- Extended locking models for entity beans (optimistic locking mode / pessimistic locking mode / read-only mode)
- Active Components for Java (AC4J), to provide a standards-based infrastructure for coordinating long-running business transactions

See the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for information.

OC4J Standalone

In a production environment, it is typical to use OC4J inside a complete Oracle Application Server environment, including the Oracle HTTP Server (as described in "Oracle HTTP Server and mod_oc4j" on page 2-2), OracleAS Web Cache, and Enterprise Manager.

For a development environment, OC4J is also available as a standalone component by downloading `OC4J_extended.zip` from the Oracle Technology Network at the following location:

```
http://www.oracle.com/technology/tech/java/oc4j
```

When using OC4J standalone, you can use its own HTTP Web listener through port 8888. For information about OC4J standalone, see the *Oracle Application Server Containers for J2EE Stand Alone User's Guide* (downloadable with OC4J_extended.zip) and the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

> **Note:** To use OC4J standalone, you must have a supported version of the Sun Microsystems JDK installed. A JDK is not provided with the OC4J standalone product.

Overview of the JSP Implementation in OC4J

The JSP container in Oracle Application Server 10*g* Release 2 (10.1.2) is compliant with the JSP 1.2 specification.

In general, a JSP 1.2 environment requires a servlet 2.3 environment, such as the OC4J servlet container. The OC4J JSP implementation, however, also supports running on a servlet 2.0 environment. To make this possible, the OC4J JSP container emulates required servlet features beyond the 2.0 specification.

For a variety of reasons, though, it is generally advisable to use the OC4J servlet 2.3 environment.

These features are discussed in the following sections:

- History and Integration of JSP Containers
- JSP Front-End Servlet and Configuration
- OC4J JSP Features for JSP 1.2
- Configurable JSP Extensions in OC4J
- Portability Across Servlet Environments

History and Integration of JSP Containers

In Oracle9*i*AS Release 1.0.2.2, the first release to include OC4J, there were two JSP containers: 1) a container developed by Oracle and known as "OracleJSP"; 2) a container licensed from Ironflare AB and known as the "Orion JSP container".

The OracleJSP container offered several advantages, including useful value-added features and enhancements such as for globalization. The Orion container also offered advantages, including superior speed, but had disadvantages as well. It did not always exhibit standard behavior when compared to the JSP reference implementation (Tomcat), and its support for internationalization and globalization was not as complete.

Oracle9*i*AS Release 2 (9.0.2) first integrated the OracleJSP and Orion containers into a single JSP container referred to in this manual as the "OC4J JSP container". This container offers the best features of both previous versions, runs efficiently as a servlet in the OC4J servlet container, and is integrated with other OC4J containers as well. The integrated container primarily consists of the OracleJSP translator and the Orion container runtime, running with a simplified dispatcher and the OC4J core runtime classes.

JSP Front-End Servlet and Configuration

The JSP container in OC4J uses the front-end servlet `oracle.jsp.runtimev2.JspServlet`. See "JSP Configuration in OC4J" on page 3-10.

For non-OC4J environments, use the old front-end servlet, `oracle.jsp.JspServlet`.

OC4J JSP Features for JSP 1.2

In the OC4J 10.1.2 implementation, the OC4J JSP container is fully compliant with the JSP 1.2 specification. Most functionality introduced in this specification is in the area of custom tag libraries.

- Tag library features:
  - There is a tag handler interface that allows iteration through a tag body without having to maintain and access a body content object.
  - You can create a tag-library-validator class and associate it with a tag library. A validator instance will check any JSP page that uses the library, to verify that it meets whatever constraints you desire.
  - For convenience, you can declare servlet event listeners in a tag library descriptor file instead of in the `web.xml` file. This enables you to more conveniently manage application and session resources associated with usage of the tag library.
  - You can package multiple tag libraries and their TLD files inside a single JAR file.

  See Chapter 8, "JSP Tag Libraries" for details about these features, and "Overview of Tag Library Changes Between the JSP 1.1 and 1.2 Specifications" on page 8-3 for a more detailed summary.
- XML features:
  - The OC4J JSP container previously supported a standard XML-alternative syntax, but this is replaced with newer technology according to the current JSP specification.
  - The OC4J JSP container generates an XML view of every translated page, which is a mapping to an XML document that describes the page. This view is available for use by tag-library-validator classes.

  See Chapter 5, "JSP XML Support" for information about these features.

ORCL00444382

- Character encoding features:

  The OC4J JSP implementation supports the `pageEncoding` attribute of the `page` directive. This enables you to specify a character encoding for the page source that is different than the character encoding for the response (specified in the `contentType` attribute).

  See "Content Type Settings" on page 9-1.

Configurable JSP Extensions in OC4J

In addition to JSP 1.2 compliance, the OC4J JSP container in Oracle Application Server 10*g* Release 2 (10.1.2) includes the following notable features.

Also see "Overview of Oracle Value-Added Features" on page 2-9.

The following have been supported since the OC4J 9.0.3 implementation:

- Mode switch to avoid JSP translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit

  The JSP specification mandates translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit, except for the `page` directive `import` attribute. These errors may be unwanted or inappropriate, for example, if a page and an included file both set an attribute to the same value (such as `language="java"`).

  In "JSP Configuration Parameter Descriptions" on page 3-13, see the description of the `forgive_dup_dir_attr` parameter.

- Separate mode switches for XML validation of `web.xml` file and TLD files

  Validation of `web.xml` is disabled by default but can be enabled. Validation of TLD files is enabled by default but can be disabled.

  In "JSP Configuration Parameter Descriptions" on page 3-13, see the descriptions of the `xml_validate` and `no_tld_xml_validate` parameters.

- Mode flag for extra imports

  Use this to automatically import certain Java packages beyond the JSP defaults.

  In "JSP Configuration Parameter Descriptions" on page 3-13, see the description of the `extra_imports` parameter.

- "Well-known" location for sharing tag libraries

  You can specify a directory where tag library JAR files can be placed for sharing across multiple Web applications.

  In "JSP Configuration Parameter Descriptions" on page 3-13, see the description of the `well_known_taglib_loc` parameter.

- Configurable JSP timeout

  You can specify a timeout value for JSP pages, after which a page is removed from memory if it has not been requested again. In "JSP-Related OC4J Configuration Parameter Descriptions" on page 3-20, see the description of the `jsp-timeout` parameter.

The following features have been supported since the OC4J 9.0.2 implementation:

- Mode switch for automatic page retranslation and reloading

  You have a choice of: 1) running JSP pages without any automatic reloading or retranslation of JSP pages; 2) automatically reloading any page implementation

ORCL00444383

classes (but not JavaBeans or other dependency classes); or 3) automatically retranslating any JSP pages that have changed.

In "JSP Configuration Parameter Descriptions" on page 3-13, see the description of the `main_mode` parameter.

- Tag handler instance pooling

  To save time in tag handler creation and garbage collection, you can optionally enable pooling of tag handler instances. They are pooled in `application` scope. You can use different settings in different pages, or even in different sections of the same page. See "Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse" on page 8-30.

- Output mode for null output

  You can print an empty string instead of the default "null" string for null output from a JSP page.

  In "JSP-Related OC4J Configuration Parameter Descriptions" on page 3-20, see the description of the `jsp-print-null` parameter.

- Single-threaded-model JSP instance pooling

  For single-threaded (non-thread-safe) JSP pages, page instances are pooled. There is no switch for this feature—it is always enabled.

Portability Across Servlet Environments

The JSP container is provided as a component of OC4J but is portable to other environments. Because the OC4J JSP container itself emulates certain required servlet features, this portability extends to older servlet environments. (Generally, a servlet 2.3 environment is required in order to support JSP 1.2 compliance.)

The servlet 2.0 specification was limited in that it provided only a single servlet context for each Java virtual machine, instead of a servlet context for each application. The OC4J JSP servlet emulation allows a full application framework in a servlet 2.0 environment, including providing applications with distinct servlet context and HTTP session objects.

Because of this extended functionality, the OC4J JSP container is not limited by the underlying servlet environment.

Oracle JDeveloper JSP Support

Visual Java programming tools now typically support JSP coding. In particular, Oracle JDeveloper supports JSP development and includes the following features:

- Integration of the OC4J JSP container to support the full application development cycle: editing, debugging, and running JSP pages
- Debugging of deployed JSP pages
- An extensive set of data-enabled and Web-enabled JavaBeans, known as JDeveloper Web beans
- The JSP Element Wizard, which offers a convenient way to add predefined Web beans to a page
- Support for incorporating custom JavaBeans
- A deployment option for JSP applications that rely on Oracle ADF Business Components, offered with JDeveloper

ORCL00444384

See "Application Deployment with Oracle JDeveloper" on page 7-25 for more information about JSP deployment support.

For debugging, JDeveloper can set breakpoints within JSP page source and can follow calls from JSP pages into JavaBeans. This is much more convenient than manual debugging techniques, such as adding print statements within the JSP page to output state into the response stream (for viewing in your browser) or to the server log (through the `log()` method of the implicit `application` object).

For information about JDeveloper, refer to the JDeveloper online help, or to the following site on the Oracle Technology Network:

`http://www.oracle.com/technology/products/jdev/content.html`

(You will need an Oracle Technology Network membership, which is free of charge.) For an overview of JSP tag libraries provided with JDeveloper, see the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Note: Other key IDE vendors have built plug-in modules that allow seamless integration with OC4J. This provides developers with the capability to build, deploy, and debug J2EE applications running on OC4J directly from within the IDE. You can refer to the following Web site for more information:

`http://www.oracle.com/technology/products/ias/oracleas_partners.html`

Overview of Oracle Value-Added Features

OC4J value-added features for JSP pages can be grouped into three major categories:

- Features implemented through custom tag libraries, custom JavaBeans, or custom classes that are generally portable to other JSP environments
- Features that are Oracle-specific
- Features supporting caching technologies

The rest of this section provides feature summaries and overviews in these areas, plus a brief summary of Oracle support for the JavaServer Pages Standard Tag Library (JSTL). JSTL support is summarized more fully in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Tag Libraries and Utilities Provided with OC4J

This section lists extended OC4J JSP features that are implemented through standards-compliant custom tag libraries, custom JavaBeans, and other classes. These features are documented in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*:

- Extended types implemented as JavaBeans that can have a specified scope
- `JspScopeListener` for event-handling
- Integration with XML and XSL
- Data-access tag library (sometimes referred to as "SQL tags") and JavaBeans
- The JSP Markup Language (JML) custom tag library, which reduces the level of Java proficiency required for JSP development

ORCL00444385

- Personalization tags, for use with Oracle Application Server Personalization
- Web services tag library
- Tag libraries and JavaBeans for uploading files, downloading files, and sending e-mail from within an application
- EJB tag library
- Additional utility tags, such as for displaying dates and currency amounts appropriately for a specified locale

Important: The `JspScopeListener` and JML tag library are deprecated in the OC4J 10.1.2 implementation and will be desupported in future implementations.

Note: See "Overview of Tags and API for Caching Support" on page 2-11 for an overview of additional tag libraries for caching.

Overview of Oracle-Specific Features

This section provides an overview of Oracle-specific programming extensions supported by the OC4J JSP container:

- *Global includes*, a mechanism to automatically statically include a file or files in multiple pages
- Dynamic Monitoring Service (DMS) support for performance measurements
- Enhanced application framework and globalization support for servlet 2.0 environments

Global Includes

The OC4J JSP container provides a feature called *global includes*. You can use this feature to specify one or more files to statically include into JSP pages in or under a specified directory, through virtual JSP `include` directives. During translation, the JSP container looks for a configuration file, `/WEB-INF/ojsp-global-include.xml`, that specifies the included files and the directories for the pages.

This enhancement is particularly useful in migrating applications that had used `globals.jsa` or `translate_params` functionality in previous Oracle JSP releases. For more information, see "Oracle JSP Global Includes" on page 7-6.

Support for Dynamic Monitoring Service

DMS adds performance-monitoring features to a number of Oracle Application Server components, including OC4J. The goal of DMS is to provide information about runtime behavior through built-in performance measurements so that users can diagnose, analyze, and debug any performance problems. DMS provides this information in a package that can be used at any time, including during live deployment. Data are published through HTTP and can be viewed with a browser.

The OC4J JSP container supports DMS features, calculating relevant statistics and providing information to DMS servlets such as the spy servlet and monitoring agent. Statistics include the following (using averages, maximums, and minimums, as applicable). Times are in milliseconds.

ORCL00444386

- Processing time of HTTP request
- Processing time of JSP service method
- Number of JSP instances created or available
- Number of active JSP instances

(Counts of JSP instances are applicable only for single-threaded situations, where `isThreadSafe` is set to `false` in a `page` directive.)

Standard configuration for these servlets is in the OC4J `application.xml` and `default-web-site.xml` configuration file. Use the Enterprise Manager to access DMS, display DMS information, and, if appropriate, alter DMS configuration.

Also see the *Oracle Application Server Performance Guide*, which contains precise definitions of the JSP metrics and detailed instructions for viewing and analyzing them.

Enhanced Servlet 2.0 Support

OC4J supports special features for a servlet 2.0 environment. It is highly advisable to migrate to the OC4J servlet 2.3 environment as soon as practical, but in the meantime, be aware of the following:

- An enhanced application framework for servlet 2.0 environments
- Extended globalization support for servlet 2.0 environments

Overview of Tags and API for Caching Support

Faced with Web performance challenges, e-businesses must invest in more cost-effective technologies and services to improve the performance of their Internet sites. *Web caching*, the caching of both static and dynamic Web content, is a key technology in this area. Benefits of Web caching include performance, scalability, high availability, cost savings, and network traffic reduction.

OC4J provides the following support for Web caching technologies:

- The JESI tag library for Edge Side Includes (ESI), an XML-style markup language that allows dynamic content assembly away from the Web server

  The OracleAS Web Cache provides an ESI engine.

- A tag library and servlet API for the Web Object Cache, an application-level cache that is embedded and maintained within a Java Web application

  The Web Object Cache uses the Oracle Application Server Java Object Cache as its default repository.

These features are documented in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Support for the JavaServer Pages Standard Tag Library

With Oracle Application Server 10*g* Release 2 (10.1.2), the OC4J JSP product supports the JavaServer Pages Standard Tag Library (JSTL), as specified in the Sun Microsystems *JavaServer Pages Standard Tag Library, Version 1.0* specification.

JSTL is intended as a convenience for JSP page authors who are not familiar or not comfortable with scripting languages such as Java. Historically, scriptlets have been used in JSP pages to process dynamic data. With JSTL, the intent is for JSTL tag usage to replace the need for scriptlets.

Key JSTL features include the following:

- JSTL expression language (EL)

  The expression language further simplifies the code required to access and manipulate application data, making it possible to avoid request-time attributes as well as scriptlets.

- Core tags for expression language support, conditional logic and flow control, iterator actions, and access to URL-based resources
- Tags for XML processing, flow control, and XSLT transformations
- SQL tags for database access
- Tags for I18N-capable internationalization and formatting

  The term "I18N" refers to an internationalization standard.

Tag support is organized into four JSTL sublibraries according to these functional areas.

For a more complete summary of JSTL support, you can refer to the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*. For complete information about JSTL, refer to the specification at the following location:

`http://www.jcp.org/aboutJava/communityprocess/first/jsr052/index.html`

Note: The custom JML, XML, and data-access (SQL) tag libraries provided with OC4J pre-date JSTL and have areas of duplicate functionality. Going forward, for standards compliance, it is generally advisable to use JSTL instead of the custom libraries.

ORCL00444388

Exhibit 25

Oracle® Application Server Web Cache

Administrator's Guide

10*g* Release 2 (10.1.2)

Part No. B14046-01

December 2004

Oracle v. epicRealm (D.Del.)
Defendant's
Deposition Exhibit
Nelson - 10

ORACLE

1

Introduction to OracleAS Web Cache

This chapter describes the performance barriers faced by Web sites and introduces the technology that can provide a complete caching solution.

This chapter contains these topics:

- What Is the Big Picture for Caching?
- Oracle's Solution to Web Site Performance Issues
- How Reverse Proxy Web Caching Works
- Benefits of Web Caching
- Features of OracleAS Web Cache
- Compatibility with Oracle Application Server Components

What Is the Big Picture for Caching?

The Web-based computing model creates new performance challenges for application developers and administrators. To carry out e-business successfully, Web sites must protect against poor response time and system outages caused by peak loads. Slow performance translates into lower productivity and lost revenue.

Many administrators try to counter this problem by adding more **application Web servers** to their existing architecture. As more users access these Web sites, more and more servers will have to be added. In short, the hardware and manageability costs associated with adding servers often outweigh the benefits.

Static caches and content distribution services can provide some relief. However, these solutions are unable to serve content that is dynamically generated. Developers spend time writing custom caching solutions.

Oracle's Solution to Web Site Performance Issues

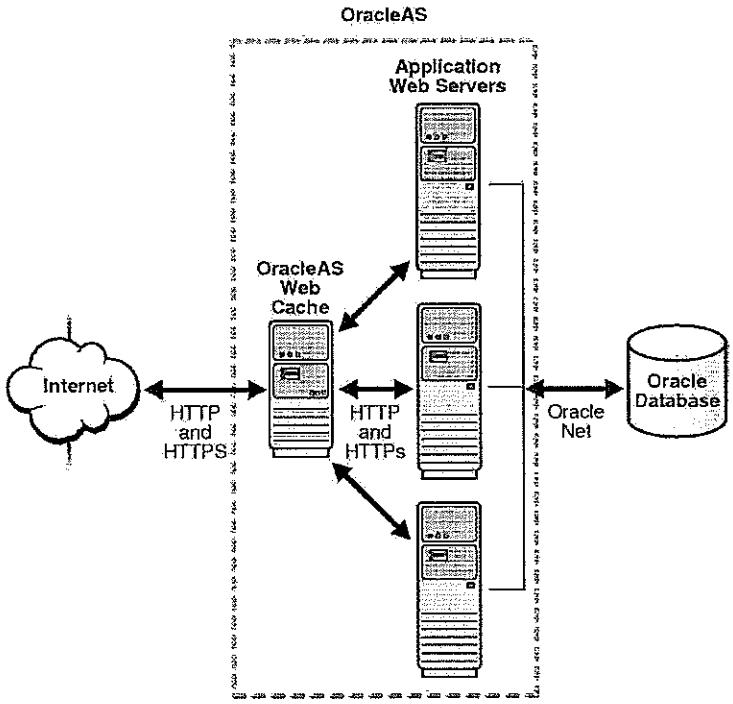
Faced with these performance challenges, e-businesses need to invest in more cost-effective technologies and services to improve the performance of their applications. Oracle offers OracleAS Web Cache to help e-businesses manage Web site and Web-based application performance issues. OracleAS Web Cache is a content-aware server accelerator, or **reverse proxy server**, that improves the performance, scalability, and availability of Web sites that run on Oracle Application Server.

By storing frequently accessed URLs in memory, OracleAS Web Cache eliminates the need to repeatedly process requests for those URLs on the application Web server and database tiers. Unlike legacy proxies that handle only static objects, OracleAS Web

Cache caches both static and dynamically generated content from one or more application Web servers. Because OracleAS Web Cache is able to cache more content than legacy proxies, it provides optimal performance by greatly reducing the load on application Web server and database tiers. As an external cache, OracleAS Web Cache is also an order of magnitude faster than object caches that run within the application tier.

Figure 1–1 shows the basic architecture. OracleAS Web Cache sits in front of application Web servers, caching their content, and providing that content to Web browsers that request it. When Web browsers access the Web site, they send **HTTP protocol** or **HTTPS protocol** requests to OracleAS Web Cache. OracleAS Web Cache, in turn, acts as a virtual server on behalf of the application Web servers. If the requested content has changed, OracleAS Web Cache retrieves the new content from the application Web servers. The application Web servers may retrieve their content from an Oracle database. OracleAS Web Cache can be deployed on its own dedicated tier of computers or on the same computer as the application Web servers.

Figure 1–1 OracleAS Web Cache Architecture



Note: OracleAS Web Cache is compatible with Oracle HTTP Server or any other HTTP-compliant application Web server. See Appendix D for further information about third-party application Web server support.

How Reverse Proxy Web Caching Works

Figure 1–2 on page 1-3 shows how reverse proxy Web caching works. OracleAS Web Cache has an IP address of `144.25.190.241` and the application Web server has an

IP address of `144.25.190.242`. The steps for browser interaction with OracleAS Web Cache follow:

1. A browser sends a request to a Web site named `www.company.com:80`.

   This request in turn generates a request to Domain Name System (DNS) for the IP address of the Web site.

2. DNS returns the IP address of the load balancer for the site, that is, `144.25.190.240`.

3. The browser sends the request for a Web page to the load balancer. In turn, the load balancer sends the request to OracleAS Web Cache server `144.25.190.241`.

4. If the requested content is in its cache, then OracleAS Web Cache sends the content directly to the browser. This is called a **cache hit**.

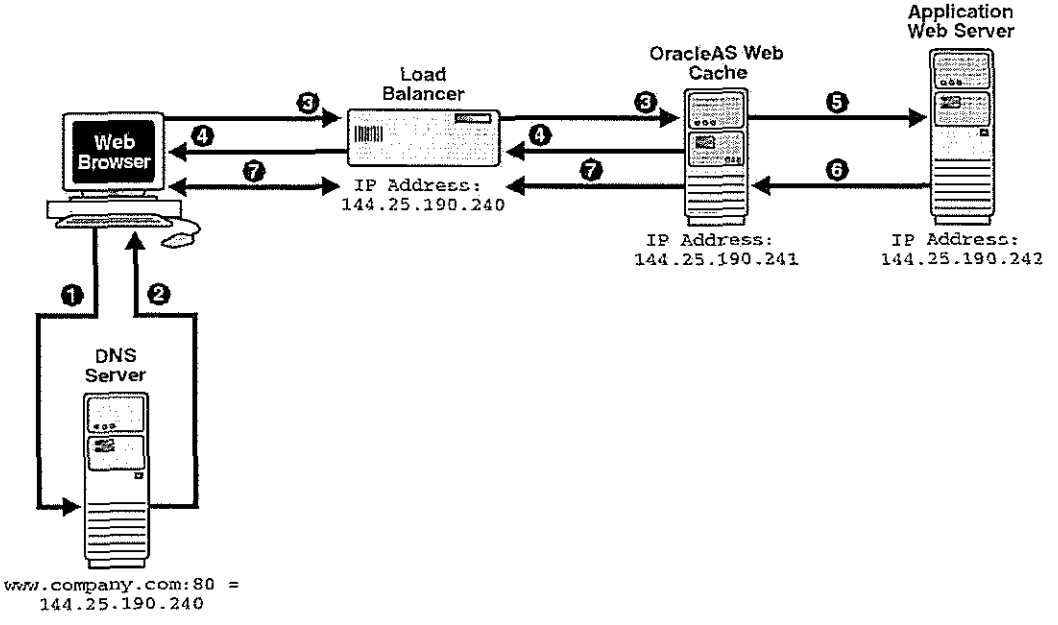
5. If OracleAS Web Cache does not have the requested content or the content is stale or invalid, it hands the request off to application Web server `144.25.190.242`. This is called a **cache miss**.

6. The application Web server sends the content to OracleAS Web Cache.

7. OracleAS Web Cache sends the content to the client and stores a copy of the page in cache.

> **Note:** A page stored in the cache is marked for removal when it becomes invalid or outdated, as described in "Cache Consistency and Performance Assurance" on page 2-2.

Figure 1–2 Web Server Acceleration



Benefits of Web Caching

Web caching provides the following benefits for Web-based applications:

- Performance

  Running on inexpensive hardware, OracleAS Web Cache can increase the throughput of a Web site by several orders of magnitude. In addition, OracleAS Web Cache significantly reduces response time to browser requests by storing objects in memory and by serving compressed versions of objects to browsers that support the GZIP encoding method.

- Scalability

  In addition to unparalleled throughput, OracleAS Web Cache can sustain thousands of concurrent browser connections, meaning that visitors to a site see fewer application Web server errors, even during periods of peak load.

- High availability

  OracleAS Web Cache supports load balancing and failover detection for application Web servers. These features ensure that cache misses are directed to the most available, highest-performing Web server in the server farm. Moreover, a patent-pending capacity heuristic guarantees performance and provides surge protection when the application Web server load increases.

- Cost savings

  Better performance, scalability and availability translates into cost savings for Web site operators. Because fewer application Web servers are required to meet the challenges posed by traffic spikes and denial of service attacks, OracleAS Web Cache offers a simple and inexpensive means of reducing a Web site's cost for each request.

- Developer productivity

  Application developers can use OracleAS Web Cache to cache content rather than design and develop application-specific caches.

Features of OracleAS Web Cache

The features of OracleAS Web Cache make it a perfect caching service for e-business Web sites that host online catalogs, news services, and portals. These features include:

- Whole-Page Caching for Static and Dynamic Content Caching
- Cache Invalidation and Expiration
- Performance Assurance
- Virtual Hosting
- Cache Clustering
- Cache Hierarchies
- Origin Server Surge Protection, Load Balancing, Failover, and Session Binding
- Security
- Compression
- Auto-Restart

Whole-Page Caching for Static and Dynamic Content Caching

Caching rules determine which objects OracleAS Web Cache caches. Rules fall into three categories:

- Rules for static content, such as GIF, JPEG, or static HTML files.
- Rules for dynamically generated content created using technologies like JavaServer Pages (JSP), Active Server Pages (ASP), PL/SQL Server Pages (PSP), Java servlets, PHP Hypertext Preprocessor (PHP), and Common Gateway Interface (CGI). Support of these technologies enables OracleAS Web Cache to recognize rules for the following:
  - Multiple-version objects for the same URL, that is, the same URL with slightly different content
  - Session-aware rules for pages containing session information
  - Personalization rules for pages containing personalized greetings, such as "Welcome *Name*," and **session-encoded URLs**
- Pages that require personalized content assembly of dynamic **Edge Side Includes (ESI)** fragments.

See Also:

- "Caching Dynamically Generated Content" on page 2-9 for further information about dynamically-generated content
- "Content Assembly and Partial Page Caching" on page 2-19

Cache Invalidation and Expiration

OracleAS Web Cache supports **invalidation** as a mechanism to keep the cache consistent with the content on the application Web servers. Administrators can invalidate cache content in one of three ways:

- Manual methods, including OracleAS Web Cache Manager or `telnet`
- Automatic methods with database triggers, scripts, or application logic
- Inline invalidation implemented as part of ESI template pages

When objects are invalidated and a browser requests them, OracleAS Web Cache refreshes them with new content from the application Web server.

In addition to invalidation, OracleAS Web Cache provides **expiration**, whereby an expiration time limit is assigned to the objects. When an object expires, OracleAS Web Cache treats it like an invalid object. If the object is requested by a browser, OracleAS Web Cache refreshes it with updated content from the application Web server.

See Also: "Cache Consistency and Performance Assurance" on page 2-2 for further information about invalidation and expiration

Performance Assurance

When a large number of objects have been invalidated, the retrieval of a new object can result in overburdened application Web servers.

To handle performance issues while maintaining cache consistency, OracleAS Web Cache uses built-in **performance assurance heuristics** that enable it to assign a queue order to objects. These heuristics determine which objects can be served stale and which objects must be refreshed immediately. Objects with a higher priority are refreshed first; objects with a lower priority are refreshed at a later time.

The queue order of objects is based on the popularity of objects and the validity of objects assigned during invalidation. If the current load and capacity of the application

Web server is not exceeded, then the most popular and least valid objects are refreshed first.

> **See Also:** "Cache Consistency and Performance Assurance" on page 2-2 for further information about performance assurance

Virtual Hosting

OracleAS Web Cache caches and assembles dynamic content for one or more Web sites. From the perspective of OracleAS Web Cache, a site can be either a **virtual host site** or an **ESI provider site**. Depending on the site category, you can configure OracleAS Web Cache to perform different functions.

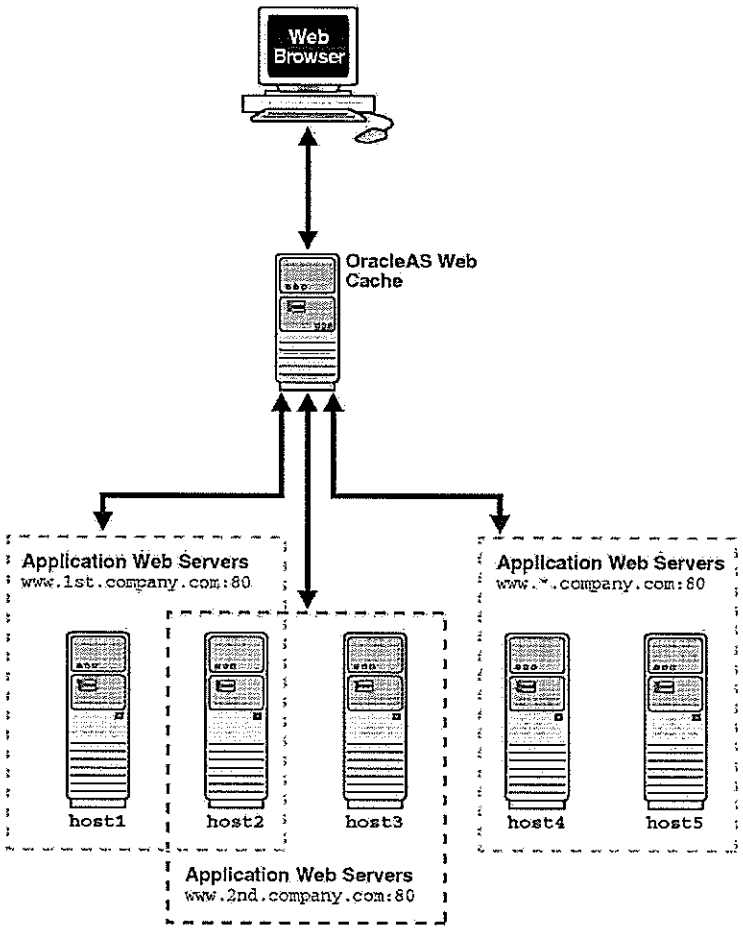
This section covers the following topics:

- Virtual Host Sites
- ESI Provider Sites
- Site Definitions and Site-to-Server Mappings
- How OracleAS Web Cache Locates Application Web Severs or Proxy Servers

Virtual Host Sites

Virtual host sites are sites hosted by OracleAS Web Cache. In other words, browsers can request cached content from these sites through OracleAS Web Cache. Figure 1–3 on page 1-7 shows OracleAS Web Cache caching content for two sites, `www.1st.company.com` and `www.2nd.company.com`. An additional mapping of `www.*.company.com` uses *, enabling additional virtual sites that map to `host4` and `host5` to be added. In addition to caching content, OracleAS Web Cache can also assemble ESI fragments from these sites.

Figure 1–3 Multiple Virtual Host Sites



ESI Provider Sites

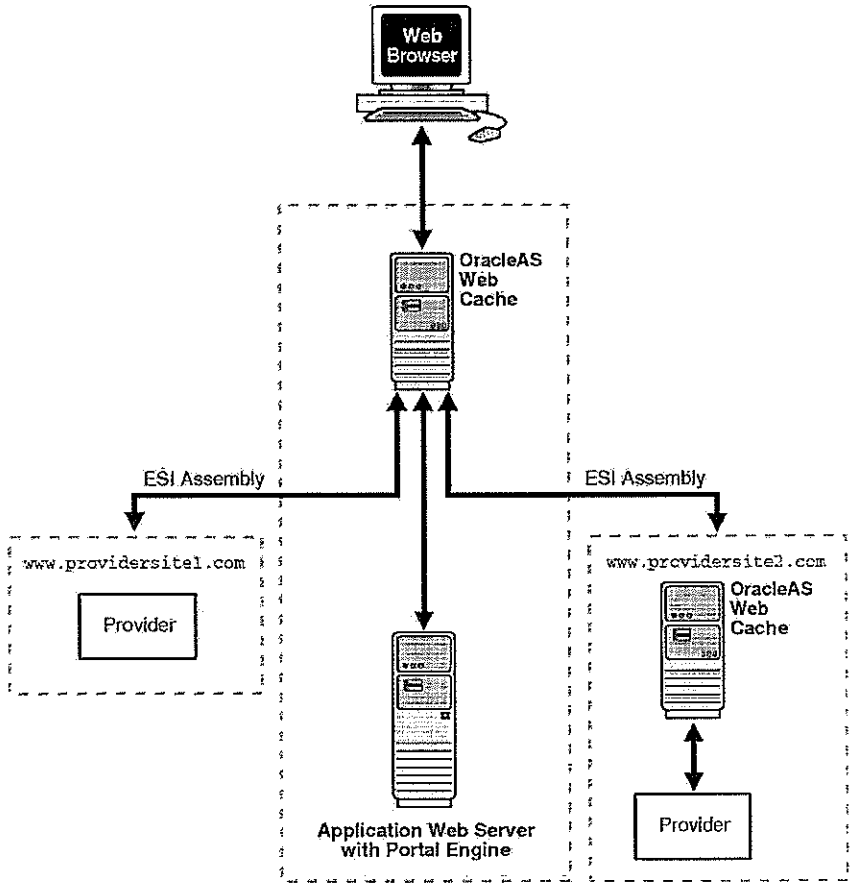
ESI provider sites are those sites that OracleAS Web Cache contacts for ESI assembly only. Browsers are not allowed to request content from these sites.

Figure 1–4 on page 1-8 shows an ESI provider site configuration. In this configuration, OracleAS Web Cache receives a request for a page with ESI markup tags. OracleAS Web Cache sends the request to the application Web server. The application Web server uses an application to create a template page and sends it back to OracleAS Web Cache for assembly. OracleAS Web Cache includes the ESI fragments for the template page either with a cached copy of the contents or by contacting the following for that fragment's contents:

- Provider site `www.providersite1.com`
- Another OracleAS Web Cache server, which is caching content for `www.providersite2.com`

> **See Also:** "Content Assembly and Partial Page Caching" on page 2-19 for further information about ESI

Figure 1–4 Multiple ESI Provider Sites



Site Definitions and Site-to-Server Mappings

In order for OracleAS Web Cache to recognize a virtual host site or an ESI provider site, administrators need to perform the following steps:

1. Create a site definition that includes all the names and listening port numbers of the site that browsers specify.

   If two sites share the same host name, specify a URL path prefix to distinguish the sites.

2. Determine if OracleAS Web Cache communicates with application Web servers or proxy servers.

   OracleAS Web Cache uses application Web servers for internal sites and proxy servers for external sites outside a firewall.

3. Specify the host name and listening port number of the application Web server or proxy server.

4. Map the site to the application Web servers or proxy servers.

5. Create caching rules that apply to the site and global rules that apply to all sites.

   The site-specific caching rules are given a higher priority than the global rules.

To further understand the mappings, reconsider Figure 1–3 on page 1-7. Web sites `www.1st.company.com:80` and `www.2nd.company.com:80` can have site aliases of `1st.company.com:80` and `2nd.company.com:80`, respectively. The site to application Web server mappings are as follows:

- `www.1st.company.com` maps to application Web servers `host1` and `host2`
- `www.2nd.company.com` maps to application Web servers `host2` and `host3`
- `www.*.company.com` maps to `host4` and `host5`

How OracleAS Web Cache Locates Application Web Severs or Proxy Servers

When OracleAS Web Cache receives a browser request for an object, it determines the destination site using one of the following elements:

- `Host` request-header field from the request
- Host portion of the requested URL
- `src` attribute of the ESI `<esi:include>` tag

OracleAS Web Cache then looks up the configured site settings and mappings to determine if the site is supported, and the origin servers and caching rules for the site. If there are no site settings and mappings for external ESI provider sites, OracleAS Web Cache uses Domain Name System (DNS) to resolve the site name.

If the request does not include host information, then OracleAS Web Cache sends the request to the default site. The default site definition includes the site host name and port information information. After an Oracle Application Server installation, a default site definition that includes the host name and listening port of the Oracle HTTP Server is established. If you select to configure another site as the default site that includes a URL path prefix, the prefix is excluded in matching requests to the site. For example, if you specify a default site of `www.company.com` and a URL path prefix of `/portal`, requests are matched for `www.company.com`, not `www.company.com/portal`.

See Also:

- Chapter 8 for configuration details
- "Default Site Settings" on page 8-36 for default site settings

Cache Clustering

To increase the availability and scalability of your Web site, you can configure multiple instances of OracleAS Web Cache to run as members of a **cache cluster**. A cache cluster is a loosely coupled collection of cooperating OracleAS Web Cache cache instances working together to provide a single logical cache.

Cache clusters provide failure detection and failover of caches, increasing the availability of your Web site. If a cache fails, other members of the cache cluster detect the failure and take over ownership of the cached content of the failed cluster member.

By distributing the Web site's content across multiple Web caches, more content can be cached and more client connections can be supported, expanding the capacity of your Web site.

See Also:

- Chapter 3 for more information about cache clusters
- Chapter 10 for configuration details

Cache Hierarchies

Many Web-based applications mirror their Web sites in strategic geographical locations. You can deploy multiple instances of OracleAS Web Cache in a **cache hierarchy** so that a local cache stores content from a cache in a central data center for a local market. This enables caches serving local requests to shorten response time and reduce bandwidth and rack space costs for the content provider.

OracleAS Web Cache provides supports two kinds of cache hierarchies:

- A **distributed cache hierarchy** in which a **central cache** acts as an application Web server to a **remote cache**.
- An **ESI cache hierarchy** in which a **provider cache** acts as an application Web server to a **subscriber cache**.

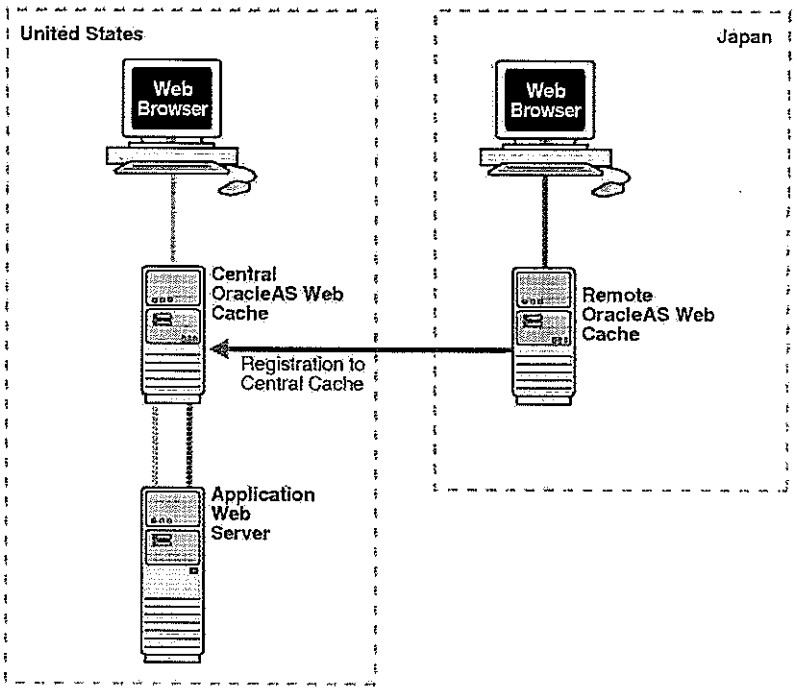
In place of an individual OracleAS Web Cache server in a hierarchy, you can deploy a cache cluster.

A distributed cache hierarchy centralizes the management of application logic and data to the central cache and provides remote assembly and delivery of content. Compared with full-scale mirroring and database replication, a distributed cache hierarchy provides a more cost-effective model of distributed computing.

The remote cache is configured with the central cache as its application Web server. When the remote cache requests content from the central cache to serve a request, the remote cache identifies itself as an OracleAS Web Cache to the central cache during a transparent registration process. After registration is complete, the central cache establishes a hierarchical relationship with the remote cache. Registration also enables invalidation messages to be propagated from the central cache to the remote cache.

Figure 1–5 depicts a distributed cache hierarchy. A central cache resides in the United States office and a remote cache resides in Japan. While the central cache in United States stored content retrieved from an application Web server, the remote cache in Japan stores retrieved content from the central cache.

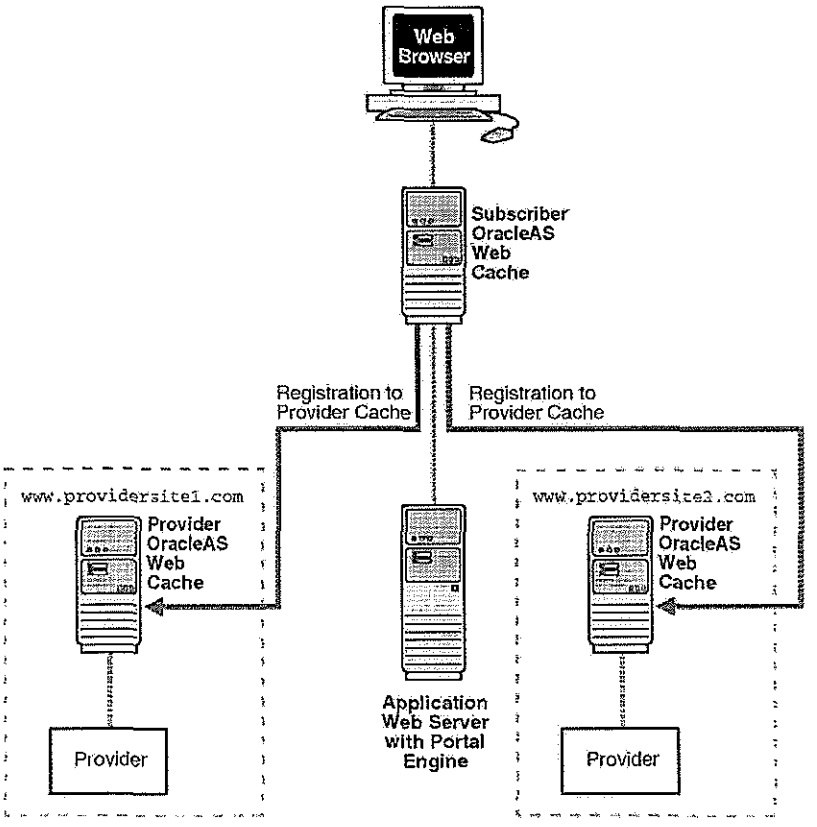
Figure 1–5 Distributed Cache Hierarchy



In an ESI cache hierarchy, the subscriber cache is configured with the provider caches as its application Web servers. When the subscriber cache requests content from the provider caches for ESI assembly, the subscriber cache identifies itself as an OracleAS Web Cache server to the provider caches during a transparent registration process. After registration is complete, the provider caches establish a hierarchical relationship with the subscriber cache. Registration also enables invalidation messages to be propagated from the provider caches to the subscriber cache.

Figure 1–6 depicts an ESI cache hierarchy. A subscriber cache performs ESI assembly. Provider caches locally store ESI fragments for **ESI provider sites** `www.providersite1.com` and `www.providersite2.com`. During ESI page assembly, the subscriber cache contacts the provider caches for the ESI fragments. By storing the ESI fragments locally on the provider caches, fragments are stored both by the provider and subscriber caches. This provides for quick page assembly.

Figure 1–6 ESI Cache Hierarchy



See Also:

- "Invalidation in Hierarchies" on page 2-3 for information about how invalidation messages are propagated in a hierarchy of OracleAS Web Cache servers
- Chapter 11 for configuration details

Origin Server Surge Protection, Load Balancing, Failover, and Session Binding

OracleAS Web Cache provides the following features for the application Web server and proxy server it supports:

- Surge Protection
- Stateless Load Balancing
- Backend Failover
- Session Binding (Stateful Load Balancing)

Where applicable, the term **origin server** is used in place of application Web server or proxy server to simplify the concepts presented in this section.

Surge Protection

OracleAS Web Cache passes requests for non-cacheable, stale, or missing objects to origin servers. To prevent an overload of requests on the origin servers, OracleAS Web Cache has a surge protection feature that enables you to set a limit on the number of

concurrent requests that the origin servers can handle. When the limit is reached, subsequent requests are queued. If the queue is full, then OracleAS Web Cache rejects the request and serves a site busy error page to the Web browser that initiated the request.

Stateless Load Balancing

Most Web sites are served by multiple origin servers running on multiple computers that share the load of HTTP and HTTPS requests. All requests that OracleAS Web Cache cannot serve are passed to the origin servers. OracleAS Web Cache balances the load among origin servers by determining the percentage of the available **capacity**, the **weighted available capacity** of each origin server. OracleAS Web Cache sends a request to the origin server with the most weighted available capacity. The weighted available capacity is determined by the following formula:

```
(Capacity - Load) / Capacity
```

where:

- `Capacity` is the maximum number of concurrent connections that the origin server can accept
- `Load` is the number of connections currently in use

If the weighted available capacity is equal for multiple origin servers, OracleAS Web Cache sends requests to the origin servers using **round robin**. With round robin, the first origin server in the list of configured servers receives the request, then the second origin server receives the second request. If the weighted available capacity is not equal, OracleAS Web Cache sends the request to the origin server with the most available capacity.

If the load of origin servers is equivalent, OracleAS Web Cache continues to use round robin, even when capacity is not equal for origin servers. Therefore, it is possible to see an even distribution of requests to origin server when the capacities are not configured to be the same.

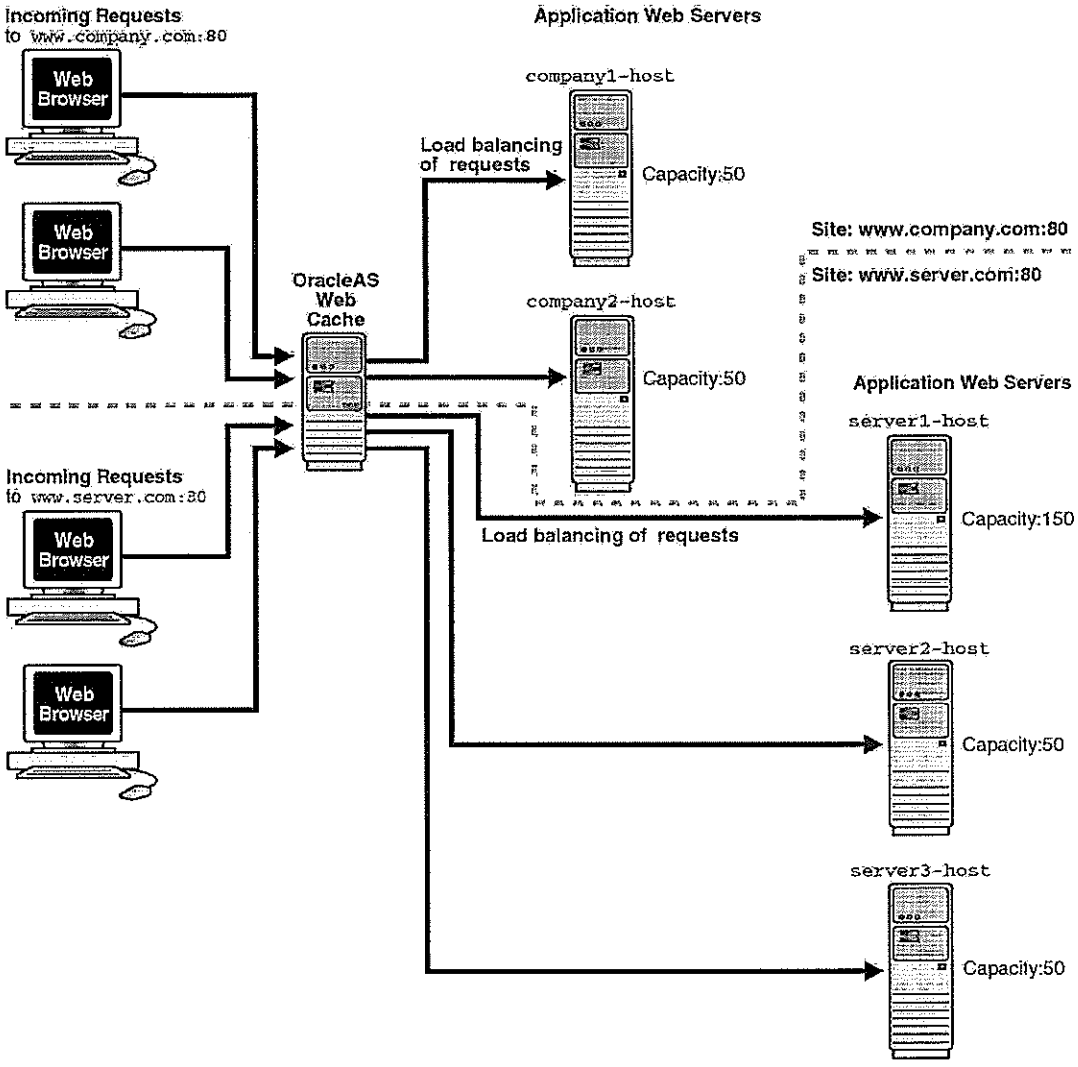
To configure load balancing for a site, set the capacity of each origin server, and create *one* site-to-server mapping that maps *all* the applicable origin servers to the site.

See Also:

- "Task 9: Configure Origin Server, Load Balancing, and Failover Settings" on page 8-23 for instructions on specifying capacity
- "Task 10: Configure Web Site Settings" on page 8-26 for instructions on creating site-to-server mappings

Figure 1–7 on page 1-14 shows two sites, `www.company.com:80` and `www.server.com:80`. The site `www.company.com:80` is supported by application Web servers `company1-host` and `company2-host` with capacities of 50 each. The site `www.server.com:80` is supported by application Web servers `server1-host`, `server2-host`, and `server3-host` with capacities of 150, 50, and 50, respectively.

Figure 1–7 Load Balancing



Assuming all application Web servers have an initial load of 0, the requests to `www.company.com:80` and `www.server.com:80` will be distributed in the following manner:

- The requests to `www.company.com:80` are distributed between the two origin servers using round robin.

  The requests to `company1-host` and `company2-host` will be distributed between the two origin servers so that they maintain an equal load. The first request is sent to `company1-host`. The second request is sent to `company2-host` if `company1-host` is still processing the first request. The third and subsequent requests are sent to the origin server that has the highest weighted available capacity.

  When the capacities are equal, OracleAS Web Cache uses round robin to distribute requests.

- The requests to `www.server.com:80` are distributed between three origin servers using the weighted available capacity percentage.

  The first request to `www.server.com:80` is sent to `server1-host`, because it is the first in the configured list. The second request is sent to `server2-host`, because `server1-host` is still processing the first request and has a weighted available capacity of 99.3 percent and `server2-host` has a weighted available capacity of 100 percent. The third request is sent to `server3-host` because `server2-host` is still processing a request and has a weighted available capacity of 98 percent and `server3-host` has a weighted available capacity of 100 percent. The fourth request is sent to `server1-host` because `server2-host` and `server3-host` are still processing requests and have weighted available capacities of 98 percent. The fifth request is sent to `server1-host` because its weighted available capacity is 98.6 percent, which is still greater than `server2-host` and `server3-host`, respectively.

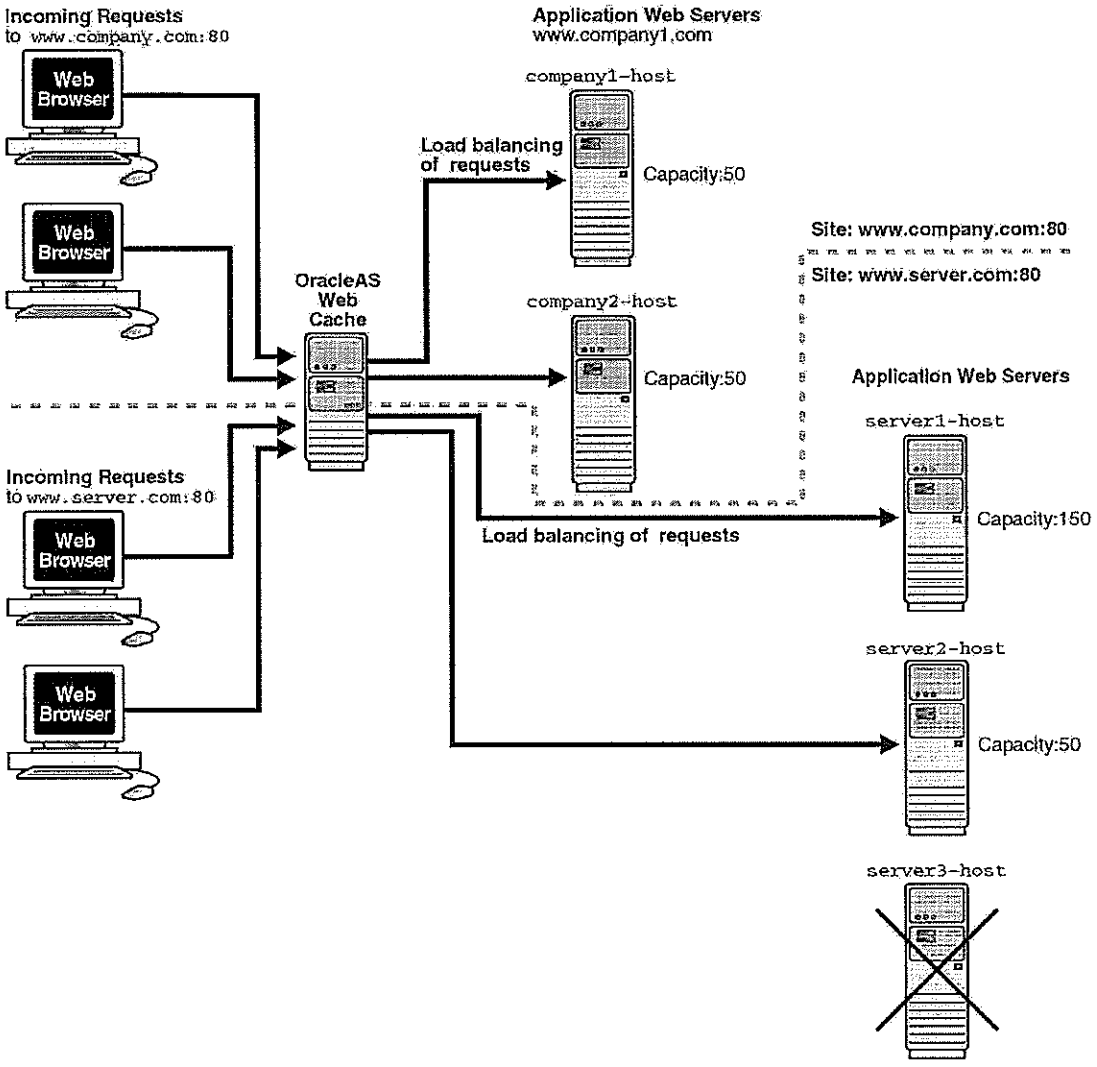
  When the capacities and loads are not equal, OracleAS Web Cache uses the weighted available capacity to distribute requests. If requests were processed before new requests came in, then it is possible for all three origin servers to have loads of 0. In this case, OracleAS Web Cache uses round robin.

Backend Failover

After a specified number of continuous request failures, OracleAS Web Cache considers an origin server as failed. When an origin server fails, OracleAS Web Cache automatically distributes the load over the remaining origin servers and polls the failed origin server for its current up or down status until it is back online. Existing requests to the failed origin server result in errors. However, new requests are directed to the other origin servers. When the failed server returns to operation, OracleAS Web Cache includes it in its weighted available capacity to load balance requests.

The **failover** feature is shown in Figure 1–8 on page 1-16. An outage of `server3-host`, which had a capacity of 50, results in 75 percent of requests being distributed to `server1-host` and 25 percent request being distributed to `server2-host`.

Figure 1–8 Failover



Session Binding (Stateful Load Balancing)

OracleAS Web Cache supports applications that use a session ID or **session cookie** to bind user sessions to a given origin server in order to maintain state for a period of time. To utilize the **session binding** feature, the origin server itself must maintain state, that is, it must be stateful. An application binds user sessions by including session data in the HTTP header or body it sends to Web browsers in such a way that the browser is forced to include it with its next request. This data is transferred between the origin server and the browser through OracleAS Web Cache either with an **embedded URL parameter** or through a cookie, which is a text string that is sent to and stored on the client browser. OracleAS Web Cache does not process the value of the parameter or cookie; it simply passes the information back and forth between the origin server and browser.

Figure 1–9 on page 1-18 shows how OracleAS Web Cache supports objects that use session binding:

1. When a request first comes in, OracleAS Web Cache uses load balancing to determine to which origin server the request is forwarded. In this example, application Web server `www.server2.com` is selected.
2. If the requested object requires session binding, the origin server sends the session information back to the browser through OracleAS Web Cache in the form of a cookie or an embedded URL parameter.
3. OracleAS Web Cache sends subsequent requests for the session to the origin server that established the session, bypassing load balancing. In this example, application Web server `www.server2.com` handles the subsequent requests.

To configure session binding:

1. Specify a session definition that specifies the name of session cookie or embedded URL parameter.

   You can also select to use one of the default definitions, described in row Session Definitions of Table 8–1 on page 8-1.

2. Specify session binding settings.

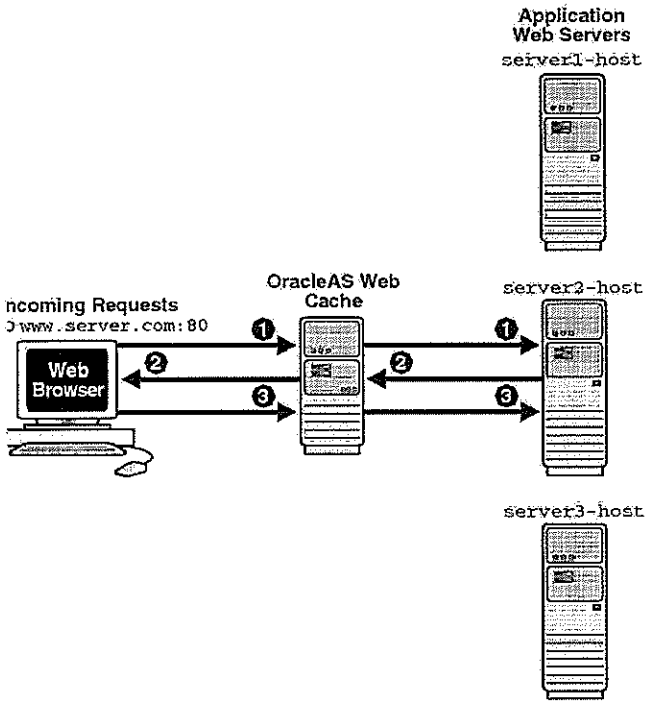
In a cache cluster, all cache cluster members must be able to determine which origin server established the session, although the request was routed originally through only one cache cluster member. To configure session binding in a cache cluster, you must enable the Session Binding cookie, which is specific to OracleAS Web Cache. OracleAS Web Cache generates the cookie, which tracks session information so that subsequent requests are bound to the origin server that established the session.

> **See Also:** "Bind a Session to an Origin Server" on page 8-33 for instructions on configuring session binding

---

Note: If an origin server cannot accept any more connections because of the load, OracleAS Web Cache disables session binding to that origin server and attempts to connect to another origin server.

Figure 1–9 Session Binding



Security

See: Chapter 4, "OracleAS Web Cache Security"

Compression

You can specify that OracleAS Web Cache compress both cacheable and non-cacheable objects with either OracleAS Web Cache Manager or the `compress` control directive of the `Surrogate-Control` response-header field.

OracleAS Web Cache correctly handles compression of different types of content and different types of browsers. For example, OracleAS Web Cache never compresses GIF or ZIP files even if you enable compression. Similarly, OracleAS Web Cache does not compress cascading style sheets for the Netscape 4.x or Internet Explorer 5.5 browsers. With these browsers, compressed cascading style sheets can cause background attributes, such as background images, to not appear in the output.

Because compressed objects are smaller in size, they are delivered faster to browsers with fewer round-trips, reducing overall **latency**. Compressed content is then expanded by browsers that support the GZIP compression in the `Accept-Encoding` request-header field.

In a cache hierarchy, the cache closest to the browser, a remote cache in a distributed cache hierarchy or a subscriber cache in an ESI cache hierarchy, does not perform any compression actions on objects. Instead, you configure the cache closest to the origin server, a central cache in a distributed cache hierarchy or a provider cache in an ESI cache hierarchy, to compress non-cacheable objects. By configuring compression on non-cacheable responses from downstream caches, you can ensure content is delivered more effectively to browsers.

Generally, the remote cache in a distributed cache hierarchy or the subscriber cache in an ESI cache hierarchy does not forward the `Accept-Encoding` request-header field to the origin server. However, if a browser request is for a non-cacheable object, the remote or subscriber cache can send this header to the central or provider cache. In turn, these caches instead compress the object.

On average, OracleAS Web Cache is able to compress text files by a factor of 4. For example, 300 KB files are compressed down to 75 KB.

See Also:

- "Configuring Caching Rules and Rule Association" on page 12-7 for instructions on configuring compression
- "Using the Surrogate-Control Response Header as an Alternative to Caching Rules" on page 12-30 for instructions on configuring the `Surrogate-Control` response-header field

Auto-Restart

OracleAS Web Cache provides an auto-restart mechanism that checks that the cache is running and automatically restarts the cache if it is not running.

See Also: "Task 3: Configure Auto-Restart Settings" on page 8-9 for instructions on enabling or disabling auto-restart

Compatibility with Oracle Application Server Components

Table 1–1 describes OracleAS Web Cache compatibility with other Oracle Application Server components.

Table 1–1 Compatibility with Other Oracle Application Server Components

Component	Description
Oracle Business Intelligence Discoverer	OracleBI Discoverer is closely integrated with OracleAS Web Cache to improve Discoverer Viewer's overall scalability, performance, and availability. OracleBI Discoverer uses ESI `Surrogate-Control` headers to govern cacheability of other non-configured responses. Because of this integration, the load on mid-tier and database servers in OracleBI Discoverer deployments is reduced, more Discoverer Viewer users are able to access the system concurrently, and those users experience significantly better response times for workbook operations and common business intelligence queries. **See Also:** *Oracle Business Intelligence Discoverer Configuration Guide*
Oracle Application Server Portal	OracleAS Web Cache has been closely integrated with OracleAS Portal to improve its overall scalability, performance, and availability. OracleAS Portal ships with a number of pre-defined caching and invalidation policies that ensure optimal use of OracleAS Web Cache. OracleAS Web Cache controls have been built into the OracleAS Portal administrative user interface and can also be specified by content providers through the Portal Developer Kit (PDK). **See Also:** Sections 5.7, "Configuring OracleAS Web Cache Caching in OracleAS Portal" and 5.8 "Configuring OracleAS Portal to Use a Dedicated OracleAS Web Cache Instance," in *Oracle Application Server Portal Configuration Guide* for configuration details

Table 1–1 (Cont.) Compatibility with Other Oracle Application Server Components

Component	Description
Oracle Application Server Single Sign-On	Applications that use OracleAS Single Sign-On can use OracleAS Web Cache. You can configure OracleAS Web Cache to cache content for Oracle HTTP Servers running Single Sign-On partner applications. While you should not cache content from OracleAS Single Sign-On servers for security reasons, you can configure OracleAS Web Cache as a software load balancer in front of multiple Single Sign-On mid-tiers. **See Also:** ▪ "Leveraging Oracle Identity Management Infrastructure" on page 4-7 ▪ *Oracle Application Server Single Sign-On Administrator's Guide*
Oracle Application Server Wireless	OracleAS Wireless is integrated with OracleAS Web Cache to improve page rendering performance and scalability. It should be noted that OracleAS Web Cache does not understand WAP and is not used by OracleAS Wireless in the traditional sense in that the cache does not "front-end" the wireless server. Instead, the cache is used as a repository for post-transformed content; the wireless runtime determines what content needs to be inserted into the cache and when to expire content in the cache. OracleAS Web Cache, in this case, acts as a device adaptation cache rather than a reverse-proxy cache. Since markup content is cached using OracleAS Web Cache, the performance and scalability benefits are due to two factors—reduced device adaptation costs and significantly reduced adapter invocation costs. The savings in terms of device adaptation costs stem from the fact that content that can be shared across users and sessions is essentially transformed only once (for each logical device) from its Mobile XML format. Secondly, since the content is not generated every time by an adapter, the total adapter invocation cost is significantly reduced for a site that has a large subset of cacheable pages. **See Also:** *Oracle Application Server Wireless Administrator's Guide*

Exhibit 26

Oracle® Call Interface

Programmer's Guide,

11*g* Release 1 (11.1)

B28395-02

October 2007

ORACLE®

Oracle Call Interface Programmer's Guide, 11*g* Release 1 (11.1)

B28395-02

Copyright © 1996, 2007, Oracle. All rights reserved.

Primary Author: Jack Melnick

Contributors: D. Alpern, Geeta Arora, Varun Arora, A. Bande, D. Banerjee, S. Banerjee, M. Bastawala, E. Belden, N. Bhatt, R. Chakravarthula, S. Chandrasekar, D. Chiba, L. Chidambaran, N. Ikeda, K. Itikarlapalli, Chandrasekhar Iyer, Shankar Iyer, B. Khaladkar, S. Kotsovolos, S. Krishnaswamy, S. Lari, Chao Liang, S. Lynn, A. Mullick, Valarie Moore, K. Neel, E. Paapanen, R. Pingte, R. Rajamani, M. Ramacher, A. Ramappa, A. Saxena, S. Seshadri, Rupa Singh, B. Sinha, H. Slattery, S. Vemuri, G. Viswanathan, S. Youssef

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

9

OCI Programming Advanced Topics

This chapter contains these topics:

- Connection Pooling in OCI
- Session Pooling in OCI
- Database Resident Connection Pooling
- When to Use Connection Pooling, Session Pooling, or Neither
- Statement Caching in OCI
- User-Defined Callback Functions in OCI
- Transparent Application Failover in OCI
- HA Event Notification
- OCI and Streams Advanced Queuing
- Publish-Subscribe Notification in OCI

Connection Pooling in OCI

Connection pooling is the use of a group (the pool) of reusable physical connections by several sessions, in order to balance loads. The management of the pool is done by OCI, not the application. Applications that can use connection pooling include middle-tier applications for Web application servers and e-mail servers.

A sample usage of this feature is in a Web application server connected to a back-end Oracle database. Suppose that a Web application server gets several concurrent requests for data from the database server. The application can create a pool (or a set of pools) in each environment during application initialization.

OCI Connection Pooling Concepts

Oracle has several transaction monitor capabilities such as the fine-grained management of database sessions and connections. This is done by separating the notion of database sessions (user handles) from connections (server handles). Using these OCI calls for session switching and session migration, it is possible for an application server or transaction monitor to multiplex several sessions over fewer physical connections, thus achieving a high degree of scalability by pooling of connections and back-end Oracle server processes.

The connection pool itself is normally configured with a shared pool of physical connections, translating to a back-end server pool containing an identical number of dedicated server processes.

The number of physical connections is less than the number of database sessions in use by the application. The number of physical connections and back-end server processes are also reduced by using connection pooling. Thus many more database sessions can be multiplexed.

Similarities and Differences from Shared Server

Connection pooling on the middle-tier is similar to what shared server offers on the back end. Connection pooling makes a dedicated server instance behave like a shared server instance by managing the session multiplexing logic on the middle tier.

The pooling of dedicated server processes including incoming connections into the dedicated server processes is controlled by the connection pool on the middle tier. The main difference between connection pooling and a shared server is that in the latter case, the connection from the client is normally to a dispatcher in the database instance. The dispatcher is responsible for directing the client request to an appropriate shared server. On the other hand, the physical connection from the connection pool is established directly from the middle-tier to the dedicated server process in the back-end server pool.

Connection pooling is beneficial only if the middle tier itself is multithreaded. Each thread can maintain a session to the database. The actual connections to the database are maintained by the connection pool and these connections (including the pool of dedicated database server processes) are shared among all the threads in the middle tier.

Stateless Sessions Versus Stateful Sessions

Stateless sessions are serially reusable across mid-tier threads. After a thread is done processing a database request on behalf of a three-tier user, the same database session can be reused for the purpose of a completely different request on behalf of a completely different three-tier user.

Stateful sessions to the database, on the other hand, are not serially reusable across mid-tier threads because they may have some particular state associated with a particular three-tier user. Examples of such state may be: open transactions, fetch state from a statement, PL/SQL package state, and so on. This makes the session not reusable for a different request for the duration that such state persists.

Note: Stateless sessions too may have open transactions, open statement fetch state, and so on. However, such a state persists for a relatively short duration (only during the processing of a particular three-tier request by a mid-tier thread) which allows the session to be serially reused for a different three-tier user (when such state is cleaned up).

Note: Stateless sessions are typically used in conjunction with Statement Caching.

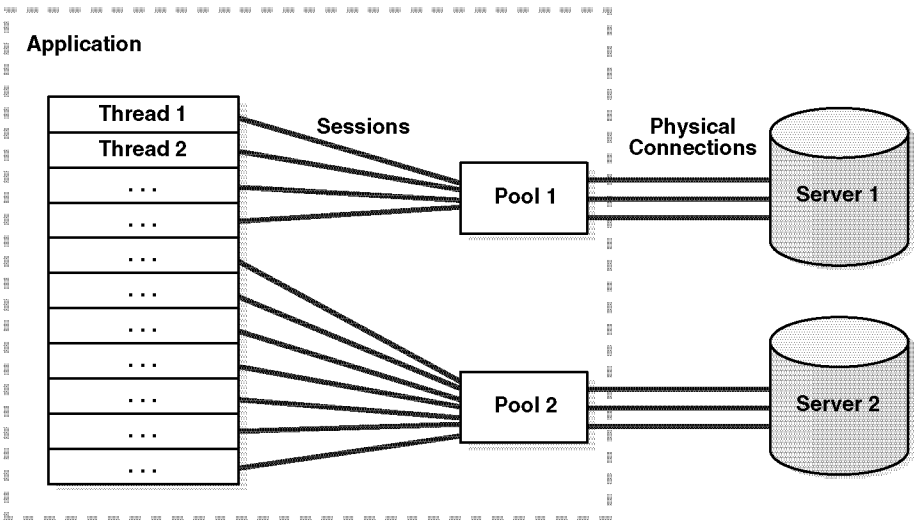
What connection pooling offers is stateless connections and stateful sessions. Users who need to work with stateless sessions, see "Session Pooling in OCI" on page 9-7.

Multiple Connection Pools

This advanced concept can be used for different database connections. Multiple connection pools can also be used when different priorities are assigned to users. Different service level guarantees can be implemented using connection pooling.

Figure 9–1, "OCI Connection Pooling" illustrates connection pooling:

Figure 9–1 OCI Connection Pooling



Transparent Application Failover

Transaction Application Failover (TAF) is enabled for connection pooling. The concepts of TAF apply equally well with connections in the connection pool except that the `BACKUP` and `PRECONNECT` clauses should not be used in the connect string and do not work with connection pooling and TAF.

When a connection in the connection pool fails over, it uses the primary connect string itself to connect. Sessions failover when they use the pool for a database round trip after their instance failure. The listener would be configured to route it to a good instance if available, as is typical with service-based connect strings.

> **See Also:** *Oracle Database Net Services Administrator's Guide*, the chapter on Configuring Transparent Application Failover

OCI Calls for Connection Pooling

The steps in using connection pooling in your application are:

- Allocate the Pool Handle
- Create the Connection Pool
- Logon to the Database
- Deal with SGA Limitations in Connection Pooling
- Logoff from the Database
- Destroy the Connection Pool
- Free the Pool Handle

Allocate the Pool Handle

Connection pooling requires that the pool handle `OCI_HTYPE_CPOOL` be allocated by `OCIHandleAlloc()`. Multiple pools can be created for a given environment handle.

For a single connection pool, here is an allocation example:

```
OCICPool *poolhp;
OCIHandleAlloc((void *) envhp, (void **) &poolhp, OCI_HTYPE_CPOOL,
                      (size_t) 0, (void **) 0));
```

Create the Connection Pool

The function OCIConnectionPoolCreate() initializes the connection pool handle. It has these IN parameters:

- connMin, the minimum number of connections to be opened when the pool is created.
- connIncr, the incremental number of connections to be opened when all the connections are busy and a call needs a connection. This increment is used only when the total number of open connections is less than the maximum number of connections that can be opened in that pool.
- connMax, the maximum number of connections that can be opened in the pool. When the maximum number of connections are open in the pool, and all the connections are busy, if a call needs a connection, it will wait till it gets one. However, if the OCI_ATTR_CONN_NOWAIT attribute is set for the pool, an error is returned.
- A poolUsername, and a poolPasswd, to allow user sessions to transparently migrate between connections in the pool.
- In addition, an attribute OCI_ATTR_CONN_TIMEOUT, can be set to time out the connections in the pool. Connections idle for more than this time are terminated periodically, to maintain an optimum number of open connections. If this attribute is not set, then the connections are never timed out.

> **Note:** Shrinkage of the pool only occurs when there is a network round trip. If there are no operations, then the connections stay alive.

All the preceding attributes can be configured dynamically. So the application has the flexibility of reading the current load (number of open connections and number of busy connections) and tuning these attributes appropriately.

If the pool attributes (connMax, connMin, connIncr) are to be changed dynamically, OCIConnectionPoolCreate() must be called with mode set to OCI_CPOOL_REINITIALIZE.

The OUT parameters poolName and poolNameLen will contain values to be used in subsequent OCIServerAttach() and OCILogon2() calls in place of the database name and the database name length arguments.

There is no limit on the number of pools that can be created by an application. Middle tier applications can take advantage of this feature and create multiple pools to connect to the same server or to different servers, to balance the load based on the specific needs of the application.

Here is an example of this call:

```
OCIConnectionPoolCreate((OCIEnv *)envhp,
                  (OCIError *)errhp, (OCICPool *)poolhp,
                  &poolName, &poolNameLen,
                  (text *)database,strlen(database),
                  (ub4) conMin, (ub4) conMax, (ub4) conIncr,
                  (text *)pooluser,strlen(pooluser),
                  (text *)poolpasswd,strlen(poolpasswd),
                  OCI_DEFAULT));
```

Logon to the Database

The application will need to log on to the database for each thread, using one of the following interfaces.

- `OCILogon2()`

  This is the simplest interface. Use this interface when you need a simple Connection Pool connection and do not need to alter any attributes of the session handle. This interface can also be used to make proxy connections to the database.

  Here is an example using `OCILogon2()`:

```
for (i = 0; i < MAXTHREADS; ++i)
{
   OCILogon2(envhp, errhp, &svchp[i], "hr", 2, "hr", 2, poolName,
             poolNameLen, OCI_LOGON2_CPOOL));

}
```

  In order to use this interface to get a proxy connection, set the password parameter to `NULL`.

- `OCISessionGet()`

  This is the recommended interface. It gives the user the additional option of using external authentication methods, such as certificates, distinguished name, and so on. `OCISessionGet()` is the recommended uniform function call to retrieve a session.

  Here is an example using `OCISessionGet()`:

```
for (i = 0; i < MAXTHREADS; ++i)
{
        OCISessionGet(envhp, errhp, &svchp, authp,
                      (OraText *) poolName,
                      strlen(poolName), NULL, 0, NULL, NULL, NULL,
                      OCI_SESSGET_CPOOL)
 }
```

- `OCIServerAttach()` and `OCISessionBegin()`:

  Another interface can be used if the application needs to set any special attributes on the user session handle and server handle. For such a requirement, applications need to allocate all the handles (connection pool handle, server handles, session handles and service context handles).

  - Create the connection pool.
  - Call `OCIServerAttach()` with mode set to `OCI_CPOOL`.
  - Call `OCISessionBegin()` with mode set to `OCI_DEFAULT`.

The user should not set `OCI_MIGRATE` flag in the call to `OCISessionBegin()`, when the virtual server handle points to a connection pool (`OCIServerAttach()` called with `mode` set to `OCI_CPOOL`). Oracle supports passing this flag, `OCI_MIGRATE`, only for compatibility reasons. Do not use the `OCI_MIGRATE` flag, because the perception that the user gets when using a connection pool is of sessions having their own dedicated (virtual) connections which are transparently multiplexed onto real connections. Credentials can be set to `OCI_CRED_RDBMS` or `OCI_CRED_PROXY`. If the credentials are set to `OCI_CRED_PROXY`, only user name needs to be set on the session handle. (no explicit primary session needs to be created and `OCI_ATTR_MIGSESSION` need not be set).

Connection pooling does the multiplexing of a virtual server handle over physical connections transparently, hence eliminating the need for users to do so. The user gets the feeling of a session having a dedicated (virtual) connection. Since the multiplexing is done transparently to the user, users must not attempt to multiplex sessions over the virtual server handles themselves. The concepts of session migration and session switching, which require explicit multiplexing at the user level, are defunct in the case of connection pooling and should not be used.

In an OCI program, the user should create (`OCIServerAttach()` with mode set to `OCI_CPOOL`) a unique virtual server handle for each session that is created using the connection pool. There should be a one-to-one mapping between virtual server handles and sessions.

Deal with SGA Limitations in Connection Pooling

With `OCI_CPOOL` mode (connection pooling), the session memory (UGA) in the back-end database will come out of the SGA. This may require some SGA tuning on the back-end database to have a larger SGA if your application consumes more session memory than the SGA can accommodate. The memory tuning requirements for the back-end database will be similar to configuring the LARGE POOL in case of a shared server back end except that the instance is still in dedicated mode.

> **See Also:** *Oracle Database Performance Tuning Guide* for more information, see the section on configuring Shared Server

If you are still running into the SGA limitation, you must consider:

- Reducing the session memory consumption by having fewer open statements for each session
- reducing the number of sessions in the back end by pooling sessions on the mid-tier or otherwise
- turning off connection pooling

The application must avoid using dedicated database links on the back end with connection pooling.

If the back end is a dedicated server, effective connection pooling will not be possible because sessions using dedicated database links will be tied to a physical connection rendering that same connection unusable by other sessions. If your application uses dedicated database links and you do not see effective sharing of back-end processes among your sessions, you must consider using shared database links.

> **See Also:** For more information about distributed databases, see the section on shared database links in *Oracle Database Administrator's Guide*

Logoff from the Database

Corresponding to the logon calls, these are the interfaces to use to log off from the database in connection pooling mode.

- `OCILogoff()`:

  If `OCILogon2()` was used to make the connection, `OCILogoff()` must be used to log off.

- `OCISessionRelease()`

  If `OCISessionGet()` was called to make the connection, then `OCISessionRelease()` must be called to log off.

- `OCISessionEnd()` and `OCIServerDetach()`

  If `OCIServerAttach()` and `OCISessionBegin()` were called to make the connection and start up the session, then `OCISessionEnd()` must be called to end the session and `OCIServerDetach()` must be called to release the connection.

Destroy the Connection Pool

Use OCIConnectionPoolDestroy() to destroy the connection pool.

Free the Pool Handle

The pool handle is freed using `OCIHandleFree()`.

These last three actions are illustrated in this code fragment:

```
for (i = 0; i < MAXTHREADS; ++i)
 {
   checkerr(errhp, OCILogoff((void *) svchp[i], errhp));
 }
 checkerr(errhp, OCIConnectionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
 checkerr(errhp, OCIHandleFree((void *)poolhp, OCI_HTYPE_CPOOL));
```

See Also:

- "Connection Pool Handle Attributes" on page A-20
- "OCIConnectionPoolCreate()" on page 16-7, "OCILogon2()" on page 16-27, and "OCIConnectionPoolDestroy()" on page 16-9

Examples of OCI Connection Pooling

Examples of connection pooling in tested complete programs can be found in `cdemocp.c and cdemocpproxy.c` in directory `demo`.

Session Pooling in OCI

Session pooling means that the application will create and maintain a group of stateless sessions to the database. These sessions will be handed over to thin clients as requested. If no sessions are available, a new one may be created. When the client is done with the session, the client will release it to the pool. Thus, the number of sessions in the pool can increase dynamically.

Some of the sessions in the pool may be 'tagged' with certain properties. For instance, a user may request for a default session, set certain attributes on it, then label it or 'tag' it and return in to the pool. That user, or some other user, can require a session with the same attributes, and thus request for a session with the same tag. There may be several sessions in the pool with the same tag. The 'tag' on a session can be changed or reset.

See Also: "Using Tags in Session Pools" on page 9-8

Proxy sessions, too, can be created and maintained through this interface.

The behavior of the application when no free sessions are available and the pool has reached it's maximum size, will depend on certain attributes. A new session may be created or an error returned, or the thread may just block and wait for a session to become free.

The main benefit of this type of pooling will be performance. Making a connection to the database is a time-consuming activity, especially when the database is remote. Thus, instead of a client spending time connecting to the server, authenticating its credentials, and then receiving a valid session, it can just pick one from the pool.

Functionality of OCI Session Pooling

Session pooling has the following features:

- Create, maintain and manage a pool of stateless sessions transparently.
- Provide an interface for the application to create a pool and specify the minimum, increment and maximum number of sessions in the pool.
- Provide an interface for the user to obtain and release a default or 'tagged' session to the pool. A 'tagged' session is one with certain client-defined properties.
- Allow the application to dynamically change the number of minimum and maximum number of sessions.
- Provide a mechanism to always maintain an optimum number of open sessions, by closing sessions that have been idle for very long, and creating sessions when required.
- Allow for session pooling with authentication.

Homogeneous and Heterogeneous Session Pools

A session pool can be either homogeneous or heterogeneous. Homogeneous session pooling means that sessions in the pool are alike with respect to authentication (have the same user name and password and privileges). Heterogeneous session pooling means that you must provide authentication information because the sessions can have different security attributes and privileges.

Using Tags in Session Pools

The tags provide a way for users to customize sessions in the pool. A client may get a default or untagged session from a pool, set certain attributes on the session (such as NLS settings), and return the session to the pool, labeling it with an appropriate tag in the `OCISessionRelease()` call.

The user, or some other user, may request a session with the same tags in order to have a session with the same attributes, and can do so by providing the same tag in the `OCISessionGet()` call.

> **See Also:** "OCISessionGet()" on page 16-37 for a further discussion of tagging sessions.

OCI Handles for Session Pooling

Two handle types have been added for session pooling:

OCISPool

This is the session pool handle. It is allocated using `OCIHandleAlloc()`. It needs to be passed to `OCISessionPoolCreate()`, and `OCISessionPoolDestroy()`. It has the attribute type `OCI_HTYPE_SPOOL`.

An example of the `OCIHandleAlloc()` call follows:

```
OCISPool *spoolhp;
```

```
OCIHandleAlloc((void *) envhp, (void **) &spoolhp, OCI_HTYPE_SPOOL,
                        (size_t) 0, (void **) 0));
```

For an environment handle, multiple session pools can be created.

OCIAuthInfo

This is the authentication information handle. It is allocated using `OCIHandleAlloc()`. It is passed to `OCISessionGet()`. It supports all the attributes that are supported for user session handle. Please refer to user session handle attributes for more information. The authentication information handle has the attribute type `OCI_HTYPE_AUTHINFO`.

An example of the `OCIHandleAlloc()` call follows:

```
OCIAuthInfo *authp;
OCIHandleAlloc((void *) envhp, (void **) &authp, OCI_HTYPE_AUTHINFO,
                      (size_t) 0, (void **) 0));
```

See Also:

- "User Session Handle Attributes" on page A-13 for the attributes that belong to the authentication information handle.
- "Session Pool Handle Attributes" on page A-22 for more information about the session pooling attributes.
- "Connect, Authorize, and Initialize Functions" on page 16-3 for complete information about the functions used in session pooling.
- See "OCISessionGet()" on page 16-37 for details of the session handle attributes that can be used with this call.

Using OCI Session Pooling

The steps in writing a simple session pooling application which uses a user name and password are:

- Allocate the session pool handle using `OCIHandleAlloc()` for an `OCISPool` handle. Multiple session pools can be created for an environment handle.
- Create the session pool using OCISessionPoolCreate() with `mode` set to `OCI_DEFAULT` (for a new session pool). Refer to the function for a discussion of the other modes.
- Loop for each thread. Create the thread with a function that does the following:
- Allocate an authentication information handle of type `OCIAuthInfo` using `OCIHandleAlloc()`.
- Set the user name and password in the authentication information handle using `OCIAttrSet()`.
- Get a pooled session using OCISessionGet() with `mode` set to `OCI_SESSGET_SPOOL`.
- Perform the transaction.
- Commit or rollback the transactions.
- Release the session (logoff) with OCISessionRelease().
- Free the authentication information handle with `OCIHandleFree()`.

- End of the loop for each thread.
- Destroy the session pool using OCISessionPoolDestroy().

OCI Calls for Session Pooling

Here are the usages for OCI calls for session pooling.

Allocate the Pool Handle

Session pooling requires that the pool handle `OCI_HTYPE_SPOOL` be allocated by calling `OCIHandleAlloc()`.

Multiple pools can be created for a given environment handle. For a single session pool, here is an allocation example:

```
OCISPool *poolhp;
OCIHandleAlloc((void *) envhp, (void **) &poolhp, OCI_HTYPE_SPOOL, (size_t) 0,
               (void **) 0));
```

Create the Pool Session

The function `OCISessionPoolCreate()` can be used to create the session pool. Here is an example of how to use this call:

```
OCISessionPoolCreate(envhp, errhp, poolhp, (OraText **)&poolName,
              (ub4 *)&poolNameLen, database,
              (ub4)strlen((const signed char *)database),
              sessMin, sessMax, sessIncr,
              (OraText *)appusername,
              (ub4)strlen((const signed char *)appusername),
              (OraText *)apppassword,
              (ub4)strlen((const signed char *)apppassword),
              OCI_DEFAULT);
```

Logon to the Database

These are the interfaces that can be used to logon to the database in session pooling mode.

- `OCILogon2()`

  This is the simplest interface. However, it does not give the user the option of using tagging. Here is an example of how `OCILogon2()` can be used to log on to the database in session pooling mode:

  ```
  for (i = 0; i < MAXTHREADS; ++i)
  {
    OCILogon2(envhp, errhp, &svchp[i], "hr", 2, "hr", 2, poolName,
              poolNameLen, OCI_LOGON2_SPOOL));

  }
  ```

- `OCISessionGet()`

  This is the recommended interface. It gives the user the option of using tagging to label sessions in the pool, and thus make it easier to retrieve specific sessions. An example of using `OCISessionGet()` follows. It is taken from `cdemosp.c` in the `demo` directory.:

  ```
  OCISessionGet(envhp, errhp, &svchp, authInfop,
                (OraText *)database,strlen(database), tag,
                strlen(tag), &retTag, &retTagLen, &found,
  ```

```
                    OCI_SESSGET_SPOOL);
```

Logoff from the Database

Corresponding to the preceding logon calls, these are the interfaces to use to log off from the database in session pooling mode.

- `OCILogoff()`

  If `OCILogon2()` was used to make the connection, `OCILogoff()` must be called to log off.

- `OCISessionRelease()`

  If `OCISessionGet()` was called to make the connection, then `OCISessionRelease()` must be called to log off.

Destroy the Session Pool

`OCISessionPoolDestroy()` must be called to destroy the session pool. Here is an example of how this call can be made:

```
OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT);
```

Free the Pool Handle

`OCIHandleFree()` must be called to free the session pool handle. Here is how this call can be made:

```
OCIHandleFree((void *)poolhp, OCI_HTYPE_SPOOL);
```

Note: The application has to ensure either a commit or rollback is done before a session is released to the session pool. OCI does not reset the state of the session.

Example of OCI Session Pooling

Here is an example of session pooling in a tested, complete program:

See Also: `cdemosp.c` in directory `demo`

Runtime Connection Load Balancing

Oracle Real Application Clusters (RAC) is a database option in which a single database is hosted by multiple instances on multiple nodes. RAC's shared disk method of clustering databases increases scalability because nodes can easily be added or freed to meet current needs and improve availability because if one node fails, another can assume its workload. It adds high availability and failover capacity to the database, because all instances have access to the whole database.

Balancing of work requests occurs at two different times: at connect time and at runtime. These are referred to as connect time load balancing provided by Oracle Net Services and runtime connection load balancing. For RAC environments, session pools use service metrics received from the RAC load balancing advisory through Fast Application Notification (FAN) events to balance application session requests. The work requests coming into the session pool can be distributed across the instances of RAC offering a service, using the current service performance.

See Also:

- `cdemosp.c` in the directory `demo`
- "Database Resident Connection Pooling" on page 9-12

Connect time load balancing occurs when a session is first created by the application. It is necessary that the sessions that are part of the pool be well distributed across RAC instances, at the time they are first created. This ensures that sessions on each of the instances get a chance to execute work.

Runtime connection load balancing is basically routing work requests to sessions in a session pool that best serve the work. It comes into effect when selecting a session from an existing session pool and thus is a very frequent activity. For session pools that support services at one instance only, the first available session in the pool is adequate. When the pool supports services that span multiple instances, there is a need to distribute the work requests across instances so that the instances that are providing better service or have greater capacity get more requests.

Runtime connection load balancing is enabled by default in a release 11.1 or higher client talking to a server of 10.2 or higher. Setting the `mode` parameter to `OCI_SPC_NO_RLB` when calling `OCISessionPoolCreate()` disables runtime connection load balancing.

Receiving Load Balancing Advisory FAN Events

To receive the service metrics based on the service time the following are required:

- RAC environment with Oracle Clusterware set up and enabled.
- The application must have been linked with the threads library.
- The OCI environment must be created in `OCI_EVENTS` and `OCI_THREADED` mode.
- The service for which the session pool is created must be enabled for event notification. To enable this use `DBMS_SERVICE.MODIFY_SERVICE` procedure and then set the `AQ_HA_NOTIFICATIONS` to `TRUE` for the service used to connect to the RAC database.
- The service should be modified to set up its goal and the connection load balancing goal:

```
EXEC DBMS_SERVICE.MODIFY_SERVICE("myService",
     DBMS_SERVICE.GOAL_SERVICE_TIME,
     clb_goal => dbms_service.clb_goal_short);
```

See Also:

- "OCISessionPoolCreate()" on page 16-41
- *Oracle Real Application Clusters Administration and Deployment Guide*, "Enabling OCI Clients to Receive FAN Events" section.
- *Oracle Database PL/SQL Packages and Types Reference*, "DBMS_SERVICE"

Database Resident Connection Pooling

Database Resident Connection Pooling (DRCP) provides a connection pool in the database server for typical Web application usage scenarios where the application acquires a database connection, works on it for a relatively short duration, and then releases it. DRCP pools "dedicated" servers, which is the equivalent of a server

foreground and a database session combined and henceforth are referred to as the "pooled" servers. DRCP complements middle-tier connection pools that share connections between threads in a middle-tier process. In addition, DRCP enables sharing of database connections across middle-tier processes on the same middle-tier host and even across middle-tier hosts. This results in significant reduction in key database resources needed to support a large number of client connections, thereby reducing the database tier memory footprint and boosting the scalability of both middle-tier and database tiers. Having a pool of readily available servers also has the additional benefit of reducing the cost of creating and tearing down client connections.

DRCP is especially relevant for architectures with multi-process single threaded application servers (such as PHP/Apache) that cannot do middle-tier connection pooling. The database can still scale to tens of thousands of simultaneous connections with DRCP.

The "pooled" server model closely follows the "dedicated" model that is used to connect to Oracle by default. It just removes the overhead of dedicating a server to every connection when the server is required only for executing short transactions. DRCP allows a connection to acquire and then voluntarily relinquish the "pooled" server, for use by other "similar" connections. On being acquired by a connection, a "pooled" server essentially transforms into a "dedicated" server for that connection, until it is released back into the pool. Clients getting connections out of the database resident connection pool connect to an Oracle background known as the connection broker. The connection broker implements the pool functionality and multiplexes pooled servers among inbound connections from the client processes.

See Also:

- *Oracle Database Concepts* for the details of the DRCP architecture
- *Oracle Database Administrator's Guide* for the details of configuring the database for DRCP

Session Purity and Connection Class

There are two new settings that can be specified when obtaining a session using `OCISessionGet()`.

Session Purity

Session purity specifies whether the application wants a "brand new" session or whether the application logic is set up to reuse a "pooled" session. `OCISessionGet()` has been enhanced to take in a purity setting of `OCI_SESSGET_PURITY_NEW` or `OCI_SESSGET_PURITY_SELF`. Alternatively, you can set `OCI_ATTR_PURITY_NEW` or `OCI_ATTR_PURITY_SELF` on the `OCIAuthInfo` handle prior to calling `OCISessionGet()`. Both these methods are equivalent.

Note: When reusing a session from the pool, the NLS attributes of the server take precedence over that of the client.

For example, if the client has `NLS_LANG` set to `french_france.us7ascii` and if it is assigned a German session from the pool, the client session would be German.

Connection classes can be used to restrict sharing and to avoid the above problem.

Setting Session Purity Examples

A connection pooling application needs a NEW session.

```
/* OCIAttrSet method */

ub4 purity = OCI_ATTR_PURITY_NEW;
OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO,  &purity, sizeof (purity),
    OCI_ATTR_PURITY, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
    NULL, NULL, NULL, OCI_DEFAULT);
/* poolName is the name returned by OCISessionPoolCreate() */

/*  OCISessionGet mode method */

OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
    NULL, NULL, NULL, OCI_SESSGET_PURITY_NEW);
/* poolName is the name returned by OCISessionPoolCreate() */
```

Connection Class

Connection class defines a logical name for the type of connection required by the application. Sessions from the OCISessionPool cannot be shared by different users - A session first created for user HR is only given out to subsequent requests by user HR. The connection class setting allows for further separation between the sessions of a given user. The connection class setting lets different applications (connecting as the same database user) identify their sessions using a logical name that corresponds to the application. OCI then ensures that such sessions belonging to a particular connection class are not shared outside of the connection class.

OCI provides a new attribute on the OCIAuthInfo handle, OCI_ATTR_CONNECTION_CLASS that can be used to set the connection class. The connection class is a string attribute. OCI supports a maximum connection class length of 1024 bytes. The character '*' is a special character and is not allowed in the connection class name.

Setting the Connection Class Examples

An HRMS application needs sessions identified with the connection class HRMS.

```
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "HRDB",
    strlen("HRDB"), 0, 10, 1, "HR", strlen("HR"), "HR", strlen("HR"),
    OCI_SPC_HOMOGENEOUS);

OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO,  "HRMS", strlen ("HRMS"),
    OCI_ATTR_CONNECTION_CLASS, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
    NULL, NULL, NULL, OCI_DEFAULT);
```

A recruitment application needs sessions identified with the connection class RECMS.

```
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "HRDB",
    strlen("HRDB"), 0, 10, 1, "HR", strlen("HR"), "HR", strlen("HR"),
    OCI_SPC_HOMOGENEOUS);

OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO,  "RECMS", strlen("RECMS"),
    OCI_ATTR_CONNECTION_CLASS, errhp);
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
    NULL, NULL, NULL, OCI_DEFAULT);
```

Defaults for Session Purity and Connection Class

Table 9–1, " Defaults Used in Various Client Scenarios" illustrates the defaults used in various client scenarios:

Table 9–1 Defaults Used in Various Client Scenarios

Attribute or Setting	Application Uses OCISessionGet() from Session Pool	Other Connections are not Obtained from OCISessionPool
OCI_ATTR_PURITY	OCI_ATTR_PURITY_SELF	OCI_ATTR_PURITY_NEW
OCI_ATTR_CONNECTION_CLASS	OCI-generated globally unique name for each client-side session pool that is used as the default connection class for all connections in the OCISessionPool.	SHARED
Sharing of sessions	Sharing of sessions between threads requesting sessions from the OCISessionPool.	Sharing among all connections of a particular database using the default SHARED connection class.

Starting the Database Resident Connection Pool

The database administrator (DBA) must logon as `SYSDBA` and start the default pool, `SYS_DEFAULT_CONNECTION_POOL`, using `DBMS_CONNECTION_POOL.START_POOL` with the default settings.

For detailed information about configuring the pool, see *Oracle Database Administrator's Guide*.

Enabling Database Resident Connection Pooling

Any application can leverage the database resident connection pool by specifying `:POOLED` in the EZ Connect string or `(SERVER=POOLED)` in the TNS connect string.

Example of TNS Connect String for Leveraging DRCP

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)
     (PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)(SERVER=POOLED)))
```

Example of EZ Connect String for Leveraging DRCP

```
oraclehost.company.com:1521/books.company.com:POOLED
```

Leveraging the Scalability of DRCP in an OCI Application

We consider three types of application scenarios and explain how each benefits from DRCP:

1. Applications that do not use OCISessionPool APIs and also do not specify any connection class/purity setting or specify a purity setting of `NEW` will simply get a brand new session from the DRCP. By the same token, when the application releases a connection back to the pool, the session is not shared with other instances of the same application by default. The application, however, does get the benefit of reusing an existing pooled server process.
2. Applications that use `OCISessionGet()` outside of an OCISessionPool, and specify the connection class and set `purity=SELF` will be able to reuse both DRCP pooled server processes and sessions. However, following an `OCISessionRelease()`, OCI will terminate the connection to the connection broker. On the next `OCISessionGet()` call, the application will reconnect to the broker. Once it reconnects, the DRCP will assign a pooled server (and session)

belonging to the connection class specified. Reconnecting, however, incurs the cost of connection establishment and re-authentication. Such applications achieve better sharing of DRCP resources (processes and sessions) but do not get the benefit of caching connections to the connection broker.

3. Applications that use OCISessionPool APIs and specify the connection class and set `purity=SELF` make full use of the Database Resident Connection Pool functionality through reuse of both the pooled server process and the associated session and also getting the benefit of cached connections to the connection broker. Cached connections do not incur the cost of re-authentication on `OCISessionGet()`.

OCISessionPool APIs have been extended to interoperate with the database resident connection pool. Sessions obtained using `OCISessionGet()` from the OCISessionPool are obtained from the DRCP and sessions released to the OCISessionPool using `OCISessionRelease()` release the sessions to DRCP. The OCISessionPool also transparently keeps connections to the connection broker cached, for better performance.

Features offered by the traditional client-side OCISessionPool, such as tagging, statement caching and TAF (Transparent application failover) are supported with DRCP also.

For information on client-side session pooling APIs:

See Also:

- "OCISessionPoolCreate()" on page 16-41
- "OCISessionGet()" on page 16-37
- "OCISessionRelease()" on page 16-45
- "OCISessionPoolDestroy()" on page 16-44

Best Practices for Using DRCP

The steps to design an application that is able to leverage the full power of DRCP are very similar to the steps required for an application that uses the OCISessionPool.

The only additional step is that for best performance, when deployed to run with DRCP, it is recommended that the application specify an explicit connection class setting.

Multiple instances of the same application should specify the same connection class setting for best performance and enhanced sharing of DRCP resources. Take care to ensure that the different instances of the application can share database sessions.

Example of a Database Resident Connection Pooling Application

```
/* Assume all necessary handles are allocated */

/*   This middle-tier uses a single database user. Create a homogeneous
     client-side SP */
OCISessionPoolCreate (envhp, errhp, spoolhp, &poolName, &poolNameLen, "BOOKSDB",
    strlen("BOOKSDB"), 0, 10, 1, "SCOTT", strlen("SCOTT"), "password",
    strlen("password"), OCI_SPC_HOMOGENEOUS);

while (1)
{
   /* Process a client request */
   WaitForClientRequest();
```

```
   /* Application function */

   /* Set the Connection Class on the OCIAuthInfo handle that is passed as
      argument to OCISessionGet*/

   OCIAttrSet (authInfop, OCI_HTYPE_AUTHINFO,  "BOOKSTORE", strlen("BOOKSTORE"),
       OCI_ATTR_CONNECTION_CLASS, errhp);

   /* Purity need not be set as default is OCI_ATTR_PURITY_SELF for OCISessionPool
       connections */

   /* We can get a SCOTT session released by Mid-tier 2 */
   OCISessionGet(envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL, 0,
       NULL, NULL, NULL, OCI_DEFAULT);

   /* Database calls using the svchp obtained above  */
   OCIStmtExecute(...)

   /* This releases the pooled server on the database for reuse */
   OCISessionRelease (svchp, errhp, NULL, 0, OCI_DEFAULT);
}

/* Mid-tier is done - exiting */
OCISessionPoolDestroy (spoolhp, errhp, OCI_DEFAULT);
```

Deployment Example 1

The code immediately preceding is deployed in 10 middle tier hosts servicing the BOOKSTORE application. The database used is 11*g* (or earlier) database in dedicated server mode but with DRCP not enabled. The client side has 11*g* libraries. The connect string used is:

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)
   (PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)))
```

In this case, the application gets dedicated server connections from the database.

Deployment Example 2

The code above is deployed in 10 middle tier hosts servicing the BOOKSTORE application. DRCP is enabled on the 11*g* database. Now all the middle tier processes can leverage the pooling capabilities offered by DRCP. The database resource requirement with DRCP will be much less than what would be required with dedicated server mode. Simply change the connect string to:

```
BOOKSDB = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oraclehost.company.com)
  (PORT=1521))(CONNECT_DATA = (SERVICE_NAME=books.company.com)(SERVER=POOLED)))
```

Compatibility and Migration

An OCI application linked with 11*g* client libraries will work unaltered against a

- 11*g* database with DRCP disabled
- pre-11g database server
- 11*g* database server with DRCP enabled, when deployed with the DRCP connect string

Suitable clients benefit from enhanced scalability offered by DRCP provided that they are appropriately modified to use the OCISessionPool APIs with the connection class and purity settings as previously described.

Restrictions on Using Database Resident Connection Pooling

The following cannot be performed or used with pooled servers:

- Shut down the database.
- Stop the database resident connection pool.
- Change the password for the connected user.
- Use shared database links to connect to a database resident connection pool that is on a different instance.
- Use Advanced Security Options (ASO) such as encryption, certificates, and so on.
- Use migratable sessions on the server side, directly by using the `OCI_MIGRATE` option or indirectly using `OCIConnectionPool`.
- Use Initial Client Roles.
- Use Application Context attributes - `OCI_ATTR_APPCTX_NAME`, `OCI_ATTR_APPCTX_VALUE`, and so on.

When to Use Connection Pooling, Session Pooling, or Neither

If database sessions are not reusable by mid-tier threads (that is, they are stateful) and the number of back-end server processes may cause scaling problems on the database, use OCI connection pooling.

If database sessions are reusable by mid-tier threads (that is, they are stateless) and the number of back-end server processes may cause scaling problems on the database, use OCI session pooling.

If database sessions are not reusable by mid-tier threads (that is, they are stateful) and the number of back-end server processes will never be large enough to potentially cause any scaling issue on the database, there is no need to use any pooling mechanism.

> **Note:** Having non-pooled sessions/connections will result in tearing down and re-creation of the database session/connection for every mid-tier user request. This can cause severe scaling problems on the database side and excessive latency for the fulfillment of the request. Hence, it is strongly recommended that one of the pooling strategies be adopted for mid-tier applications based on whether the database session is stateful or stateless.

In connection pooling, the pool element is a connection and in session pooling, the pool element is a session.

As with any pool, the pooled resource is locked by the application thread for a certain duration until the thread has done its job on the database and the resource is released. The resource is unavailable to other threads during its period of use. Hence, application developers need to be aware that any kind of pooling works effectively with relatively short tasks. If the application is performing a long running transaction for example, it may deny the pooled resource to other sharers for long periods of time

leading to starvation. Hence, pooling should be used in conjunction with short tasks and the size of the pool should be sufficiently large to maintain the desired concurrency of transactions.

Also, note that with:

1. OCI Connection Pool
   a. Connections to the database are pooled. Sessions are created and destroyed by the user. Each call to the database will pick up an appropriate available connection from the pool.
   b. The application is multiplexing several sessions over fewer physical connections to the database. The users can tune the pool configuration to achieve required concurrency.
   c. The life-time of the application sessions is independent of the life-time of the cached pooled connections.
2. OCI Session Pool

   Sessions and connections are pooled by OCI. The application gets sessions from the pool and releases sessions back to the pool.

Functions for Session Creation

The choices are:

1. `OCILogon()`

   This is the simplest way to get an OCI Session. The advantage is ease of obtaining an OCI service context. The disadvantage is that you cannot perform any advance OCI operations like session migration, proxy authentication, using a connection pool, or a session pool.

   See Also: "Application Initialization, Connection, and Session Creation" on page 2-15

2. `OCILogon2()`

   This includes the functionality of `OCILogon()` to get a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The `mode` parameter value that the function is called with determines its behavior.

   The user cannot modify the attributes (except `OCI_ATTR_STMTCACHESIZE`) of the service context returned by OCI.

   See Also: "OCILogon2()" on page 16-27

3. `OCISessionBegin()`

   This supports all the various options of an OCI session such as proxy authentication, getting a session from a connection pool or a session pool, external credentials, and migratable sessions. This is the lowest level call where all handles are needed to be explicitly allocated and all attributes set, and `OCIServerAttach()` is to be called prior to this call.

   See Also: "OCISessionBegin()" on page 16-33

4. `OCISessionGet()`

   This is now the recommended method to get a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The `mode` parameter value that the function is called with determines its behavior. This works like `OCILogon2()` but additionally enables you to specify tags for obtaining specific sessions from the pool.

   See Also: "OCISessionGet()" on page 16-37

Choosing Between Different Types of OCI Sessions

The choices are:

- Basic OCI Sessions

  This works by using user name and password over a dedicated OCI server handle. This is the no-pool mechanism. See earlier notes of when to use it.

  If authentication is obtained through external credentials, then user name or password is not required.

- Session Pool Sessions

  These sessions are from the session pool cache. Some sessions may be tagged. These are stateless sessions. Each `OCISessionGet()` and `OCISessionRelease()` call gets and releases a session from the session cache. This saves the server from creating and destroying sessions.

  See the earlier notes on connection pool sessions versus session pooling sessions versus no-pooling sessions.

- Connection Pool Sessions

  These are sessions created using `OCISessionGet()` and `OCISessionBegin()` calls from an OCI Connection Pool. There is no session cache as these are stateful sessions. Each call creates a new session and the user is responsible for terminating these sessions.

  The sessions are automatically migratable between the server handles of the connection pool. Each session can have user name and password or be a proxy session. See the earlier notes on connection pool sessions versus session pooling sessions versus no-pooling sessions.

- Sessions Sharing a Server Handle

  You can multiplex several OCI sessions over a few physical connections. The application does this manually by having the same server handle for these multiple sessions. It is preferred to have the session multiplexing details be left to OCI by using the OCI Connection Pool APIs.

- Proxy Sessions

  This is useful if the password of the client needs to be protected from the middle-tier. Proxy sessions can also be part of OCI Connection Pool and OCI Session Pool.

  See Also: "Middle-Tier Applications in OCI" on page 8-10

- Migratable Sessions

With transaction handles being migratable, there should be no need for applications to use this older feature, in light of OCI Connection Pooling.

See Also: "OCI Session Management" on page 8-9

Statement Caching in OCI

Statement caching refers to the feature that provides and manages a cache of statements for each session. In the server, it means that cursors are ready to be used without the need to parse the statement again. Statement caching can be used with connection pooling and with session pooling, and will improve performance and scalability. It can be used without session pooling as well. The OCI calls that implement statement caching are:

- `OCIStmtPrepare2()`
- `OCIStmtRelease()`

Statement Caching without Session Pooling in OCI

Users perform the usual OCI steps to logon. The call to obtain a session will have a mode that specifies whether statement caching is enabled for the session. Initially the statement cache will be empty. Developers will try to find a statement in the cache using the statement text. If the statement exists the API will return a previously prepared statement handle, otherwise it will return a newly prepared statement handle.

The application developer can perform binds and defines and then simply execute and fetch the statement before returning the statement back to the cache. In the latter case, where the statement handle was not found, the developer will need to set different attributes on the handle in addition to the other steps.

`OCIStmtPrepare2()` will also take a mode which will determine if the developer wants a prepared statement handle or a null statement handle if the statement is not found in the cache.

The pseudo code will look like:

```
OCISessionBegin( userhp, ... OCI_STMT_CACHE)  ;
OCIAttrset(svchp, userhp, ...);  /* Set the user handle in the service context */
OCIStmtPrepare2(svchp, &stmthp, stmttext, key, ...);
OCIBindByPos(stmthp, ...);
OCIDefineByPos(stmthp, ...);
OCIStmtExecute(svchp, stmthp, ...);
OCIStmtFetch(svchp, ...);
OCIStmtRelease(stmthp, ...);
...
```

Statement Caching with Session Pooling in OCI

The concepts remain the same, except that the statement cache is enabled at the session pool layer rather than at the session layer.

The attribute `OCI_ATTR_SPOOL_STMTCACHESIZE` sets the default statement cache size for each of the sessions in the session pool to this value. It is set on the `OCI_HTYPE_SPOOL` handle. The statement cache size for a particular session in the pool can be overridden at any time by using `OCI_ATTR_STMTCACHESIZE` on that session. The value of `OCI_ATTR_SPOOL_STMTCACHESIZE` can be changed at any time. This attribute can be used to enable or disable statement caching at the pool level, after creation, just as attribute `OCI_ATTR_STMTCACHESIZE` (on the service

context) is used to enable or disable statement caching at the session level. This change will be reflected on individual sessions in the pool, when they are handed to a user. Tagged sessions are an exception to this behavior. This is explained later.

Enabling or disabling of statement caching is allowed on individual pooled sessions similar to non-pooled sessions.

A user can enable statement caching on a session retrieved from a non-statement cached pool in an `OCISessionGet()` or `OCILogon2()` call by specifying `OCI_SESSGET_STMTCACHE` or `OCI_LOGON2_STMTCACHE`, respectively, in the mode argument.

When a user asks for a session from a session pool, the statement cache size for that session will default to that of the pool. This may also mean enabling or disabling statement caching in that session. For example, if a pooled session (session A) has statement caching enabled, and statement caching is turned off in the pool, and a user asks for a session, and session A is returned, then statement caching will be turned off in Session A. As another example, if Session A in a pool does not have statement caching enabled, and statement caching at the pool level is turned on, then before returning session A to a user, statement caching on Session A with size equal to that of the pool is turned on.

This will not hold true if a tagged session is asked for and retrieved. In this case, the size of the statement cache will not be changed. Consequently, it will not be turned on or off. Moreover, if the user specifies mode `OCI_SESSGET_STMTCACHE` in the `OCISessionGet()` call, this will be ignored if the session is tagged. In our earlier example, if Session A was tagged, then it is returned as is to the user.

Rules for Statement Caching in OCI

Here are some rules to follow:

- Use the function `OCIStmtPrepare2()` instead of `OCIStmtPrepare()`. If you are using `OCIStmtPrepare()`, you are strongly urged not to use a statement handle across different service contexts. Doing so will raise an error if the statement has been obtained by `OCIStmtPrepare2()`. Migration of a statement handle to a new service context actually closes the cursor associated with the old session and therefore no sharing is achieved. Client-side sharing is also not obtained, because OCI will free all buffers associated with the old session when the statement handle is migrated.
- You are urged to keep one service context for each session and use statement handles only for that service context. That will be the preferred and recommended model and usage.
- A call to `OCIStmtPrepare2()`, even if the session does not have a statement cache, will also allocate the statement handle and therefore applications using only `OCIStmtPrepare2()` must not call `OCIHandleAlloc()` for the statement handle.
- A call to the `OCIStmtPrepare2()` must be followed with `OCIStmtRelease()` after the user is done with the statement handle. If statement caching is used, this will release the statement to the cache. If statement caching is not used, the statement will be deallocated. Do not call `OCIHandleFree()` to free the memory.
- If the call to `OCIStmtPrepare2()` is made with the `OCI_PREP2_CACHE_SEARCHONLY` mode and a `NULL` statement was returned (statement was not found), the subsequent call to `OCIStmtRelease()` is not required and must not be performed.

- Do not call `OCIStmtRelease()` for a statement that was prepared using `OCIStmtPrepare()`.
- The statement cache has a maximum size (number of statements) which can be modified by an attribute on the service context, `OCI_ATTR_STMTCACHESIZE`. The default value is 20. This attribute can also be used to enable or disable statement caching for the session, pooled or non-pooled. If `OCISessionBegin()` is called without mode set as `OCI_STMT_CACHE`, then `OCI_ATTR_STMTCACHESIZE` can be set on the service context to a nonzero attribute to turn on statement caching. If statement caching is not turned on at the session pool level, `OCISessionGet()` will return a non-statement cache-enabled session. `OCI_ATTR_STMTCACHESIZE` can be used to turn the caching on. Similarly the same attribute can be used to turn off statement caching by setting the cache size to zero.
- You can choose to tag a statement at the release time so that the next time you can request a statement of the same tag. The tag will be used to search the cache. An untagged statement (tag is `NULL`) is a special case of a tagged statement. Two statements are considered different if they only differ in their tags, or if one is untagged and the other is not.

  See Also: For information about the functions for statement caching,

  - "Statement Functions" on page 17-3
  - "Service Context Handle Attributes" on page A-8
  - "Session Pool Handle Attributes" on page A-22

OCI Statement Caching Code Example

Here is an example of statement caching:

See Also: Please refer to `cdemostc.c` in directory `demo` for a working example of statement caching.

User-Defined Callback Functions in OCI

The Oracle Call Interface has the capability to execute user-specific code in addition to OCI calls. This functionality can be used for:

- Adding tracing and performance measurement code to enable users to tune their applications.
- Performing pre- or post-processing code for specific OCI calls.
- Accessing other data sources with OCI by using the native OCI interface for Oracle databases and directing the OCI calls to use user callbacks for non-Oracle data sources.

The OCI callback feature has been added by providing support for calling user code before or after executing the OCI calls. Functionality has also been provided to allow the user-defined code to be executed instead of executing the OCI code.

The user callback code can also be registered dynamically without modifying the source code of the application. The dynamic registration is implemented by loading up to five user-created dynamically linked libraries after the initialization of the environment handle during the `OCIEnvCreate()` call. These user-created libraries (such as Dynamic Link Libraries (DLLs) on Windows, or shared libraries on Solaris register the user callbacks for the selected OCI calls transparently to the application.

Sample Application

For a listing of the complete demonstration programs that illustrate the OCI user callback feature, see Appendix B, "OCI Demonstration Programs".

Registering User Callbacks in OCI

An application can register user callback libraries with the `OCIUserCallbackRegister()` function. Callbacks are registered in the context of the environment handle. An application can retrieve information about callbacks registered with a handle with the `OCIUserCallbackGet()` function.

> **See Also:** "OCIUserCallbackGet()" on page 17-190 and "OCIUserCallbackRegister()" on page 17-192

A user-defined callback is a subroutine that is registered against an OCI call and an environment handle. It can be specified to be either an entry callback, a replacement callback, or an exit callback.

- If it is an entry callback, it is called when the program enters the OCI function.
- Replacement callbacks are executed after entry callbacks. If the replacement callback returns a value of `OCI_CONTINUE`, then a subsequent replacement callback or the normal OCI-specific code is executed. If a replacement callback returns anything other than `OCI_CONTINUE`, subsequent replacement callbacks and the OCI code does not execute.
- After a replacement callback returns something other than `OCI_CONTINUE`, or an OCI function successfully executes, program control transfers to the exit callback (if one is registered).

If a replacement or exit callback returns anything other than `OCI_CONTINUE`, then the return code from the callback is returned from the associated OCI call.

A user callback can return `OCI_INVALID_HANDLE` when either an invalid handle or an invalid context is passed to it.

> **Note:** If any callback returns anything other than `OCI_CONTINUE`, then that return code is passed to the subsequent callbacks. If a replacement or exit callback returns a return code other than `OCI_CONTINUE`, then the final (not `OCI_CONTINUE`) return code is returned from the OCI call.

OCIUserCallbackRegister

A user callback is registered using the OCIUserCallbackRegister() call.

> **See Also:** "OCIUserCallbackRegister()" on page 17-192

Currently, `OCIUserCallbackRegister()` is only registered on the environment handle. The user's callback function of typedef `OCIUserCallback` is registered along with its context for the OCI call identified by the OCI function code, *fcode*. The type of the callback, whether entry, replacement, or exit, is specified by the *when* parameter.

For example, the `stmtprep_entry_dyncbk_fn` entry callback function and its context `dynamic_context`, are registered against the environment handle `hndlp` for the `OCIStmtPrepare()` call by calling the `OCIUserCallbackRegister()` function with the following parameters.

```
OCIUserCallbackRegister( hndlp,
                         OCI_HTYPE_ENV,
                         errh,
                         stmtprep_entry_dyncbk_fn,
                         dynamic_context,
                         OCI_FNCODE_STMTPREPARE,
                         OCI_UCBTYPE_ENTRY
                         (OCIUcb*) NULL);
```

User Callback Function

The user callback function has to follow the following syntax:

```
typedef sword (*OCIUserCallback)
     (void *ctxp,      /* context for the user callback*/
      void *hndlp,     /* handle for the callback, env handle for now */
      ub4 type,         /* type of handlp, OCI_HTYPE_ENV for this release */
      ub4 fcode,        /* function code of the OCI call */
      ub1 when,         /* type of the callback, entry or exit */
      sword returnCode, /* OCI return code */
      ub4 *errnop,      /* Oracle error number */
      va_list arglist); /* parameters of the oci call */
```

In addition to the parameters described in the `OCIUserCallbackRegister()` call, the callback is called with the return code, *errnop*, and all the parameters of the original OCI as declared by the prototype definition.

The return code is always passed in as `OCI_SUCCESS` and `*errnop` is always passed in as 0 for the first entry callback. Note that `*errnop` refers to the content of `errnop` because `errnop` is an IN/OUT parameter.

If the callback does not want to change the OCI return code, then it must return `OCI_CONTINUE`, and the value returned in `*errnop` is ignored. If on the other hand, the callback returns any other return code than `OCI_CONTINUE`, the last returned return code becomes the return code for the call. At the this point, the value of `*errnop` returned is set in the error handle, or in the environment handle if the error information is returned in the environment handle because of the absence of the error handle for certain OCI calls such as `OCIHandleAlloc()`.

For replacement callbacks, the `returnCode` is the non-`OCI_CONTINUE` return code from the previous callback or OCI call and `*errnop` is the value of the error number being returned in the error handle. This allows the subsequent callback to change the return code or error information if needed.

The processing of replacement callbacks is different in that if it returns anything other than `OCI_CONTINUE`, then subsequent replacement callbacks and OCI code is bypassed and processing jumps to the exit callbacks.

Note that if the replacement callbacks return `OCI_CONTINUE` to allow processing of OCI code, then the return code from entry callbacks is ignored.

All the original parameters of the OCI call are passed to the callback as variable parameters and the callback must retrieve them using the *va_arg* macros. The callback demonstration programs provide examples.

See Also: Appendix B, "OCI Demonstration Programs"

A null value can be registered to de-register a callback. That is, if the value of the callback (`OCIUserCallback()`) is `NULL` in the `OCIUserCallbackRegister()` call, then the user callback is de-registered.

When using the thread-safe mode, the OCI program acquires all mutexes before calling the user callbacks.

UserCallback Control Flow

This pseudocode describes the overall processing of a typical OCI call:

```
OCIXyzCall()
{
 Acquire mutexes on handles;
 retCode = OCI_SUCCESS;
 errno = 0;
 for all ENTRY callbacks do
  {

     EntryretCode = (*entryCallback)(..., retcode, &errno, ...);
     if (retCode != OCI_CONTINUE)
      {
         set errno in error handle or environment handle;
         retCode = EntryretCode;
       }
   }
  for all REPLACEMENT callbacks do
  {
   retCode = (*replacementCallback) (..., retcode, &errno, ...);
   if (retCode != OCI_CONTINUE)
      {
       set errno in error handle or environment handle
       goto executeEXITCallback;
       }
   }

   retCode = return code for XyzCall; /* normal processing of OCI call */

   errno = error number from error handle or env handle;

 executeExitCallback:
   for all EXIT callbacks do
   {
       exitRetCode = (*exitCallback)(..., retCode, &errno,...);
       if (exitRetCode != OCI_CONTINUE)
       {
           set errno in error handle or environment handle;
           retCode = exitRetCode;
       }
   }
    release mutexes;
    return retCode
}
```

UserCallback for OCIErrorGet()

If the callbacks are a total replacement of the OCI code, then they usually maintain their own error information in the call context and use that to return error information in `bufp` and `errnop` parameters of the replacement callback of the `OCIErrorGet()` call.

If on the other hand, the callbacks are either partially overriding OCI code, or just doing some other post processing, then they can use the exit callback to modify the error text and `errnop` parameters of the `OCIErrorGet()` by their own error message

and error number. Note that the `*errnop` passed into the exit callback is the error number in the error or the environment handle.

Errors from Entry Callbacks

If an entry callback wants to return an error to the caller of the OCI call, then it must register a replacement or exit callback. This is because if the OCI code is executed, then the error code from the entry callback is ignored. Therefore the entry callback must pass the error to the replacement or exit callback through its own context.

Dynamic Callback Registrations

Because user callbacks are expected to be used for monitoring OCI behavior or to access other data sources, it is desirable that the registration of the callbacks be done transparently and non-intrusively. This is accomplished by loading user-created dynamically linked libraries at OCI initialization time. These dynamically linked libraries are called *packages*. The user-created packages register the user callbacks for the selected OCI calls. These callbacks can further register or de-register user callbacks as needed when receiving control at runtime.

A makefile (`ociucb.mk` on Solaris) is provided with the OCI demonstration programs to create the package. The exact naming and location of this package is operating system dependent. The source code for the package must provide code for special callbacks that are called at OCI initialization and environment creation times.

The loading of the package is controlled by setting an operating system environment variable, `ORA_OCI_UCBPKG`. This variable names the packages in a generic way. The packages must be located in the `$ORACLE_HOME/lib` directory.

Loading Multiple Packages

The `ORA_OCI_UCBPKG` variable can contain a semicolon separated list of package names. The packages are loaded in the order they are specified in the list.

For example, previously one specified the package as:

```
setenv ORA_OCI_UCBPKG mypkg
```

Now, you can still specify the package as earlier, but in addition multiple packages can be specified as:

```
setenv ORA_OCI_UCBPKG "mypkg;yourpkg;oraclepkg;sunpkg;msoftpkg"
```

All these packages are loaded in order. That is, `mypkg` is loaded first and `msoftpkg` is loaded last.

A maximum of five packages can be specified.

Note: The sample makefile `ociucb.mk` creates `ociucb.so.1.0` on a Solaris or `ociucb.dll` on a Windows system. To load the `ociucb` package, the environmental variable `ORA_OCI_UCBPKG` must be set to `ociucb`. On Solaris, if the package name ends with `.so`, `OCIInitialize()` fails. The package name must end with `.so.1.0`.

For further details about creating the dynamic link libraries, read the Makefiles provided in the demo directory for your operating system. For further information on user-defined callbacks, see your operating system-specific documentation on compiling and linking applications.

Package Format

Previously a package had to specify the source code for the `OCIEnvCallback()` function. Now the `OCIEnvCallback()` function is obsolete. Instead, the package source must provide two functions. The first function has to be named as *packagename* suffixed with the word *Init*. For example, if the package is named `foo`, then the source file (for example, but not necessarily, foo.c) must contain a `fooInit()` function with a call to `OCISharedLibInit()` function specified exactly as:

```
sword fooInit(metaCtx, libCtx, argfmt, argc, argv)
      void *     metaCtx;         /* The metacontext */
      void *     libCtx;          /* The context for this package. */
      ub4        argfmt;          /* package argument format */
      sword      argc;            /* package arg count*/
      void *     argv[];          /* package arguments */
{
  return  (OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv,
                            fooEnvCallback));
}
```

The last parameter of the `OCISharedLibInit()` function, `fooEnvCallback()`, in this case, is the name of the second function. It can be named anything, but by convention it can be named *packagename* suffixed with the word *EnvCallback*.

This function is a replacement for `OCIEnvCallback()`. Now all the dynamic user callbacks must be registered in this function. The function must be of type `OCIEnvCallbackType`, which is specified as:

```
typedef sword (*OCIEnvCallbackType)(OCIEnv *env, ub4 mode,
                                    size_t xtramem_sz, void *usrmemp,
                                    OCIUcb *ucbDesc);
```

When an environment handle is created, then this callback function is called at the very end. The `env` parameter is the newly created environment handle.

The `mode`, `xtramem_sz`, and `usrmemp` are the parameters passed to the `OCIEnvCreate()` call. The last parameter, `ucbDesc`, is a descriptor that is passed to the package. The package uses this descriptor to register the user callbacks as described later.

A sample `ociucb.c` file is provided in the demo directory. The makefile `ociucb.mk` is also provided (on Solaris) in the `demo` directory to create the package. Please note that this may be different on other operating systems. The demo directory also contains full user callback demo programs (`cdemoucb.c, cdemoucbl.c`,) illustrating this.

User Callback Chaining

User callbacks can both be registered statically in the application itself or dynamically at runtime in the DLLs. A mechanism is needed to allow the application to override a previously registered callback and then later invoke the overridden one in the newly registered callback to preserve the behavior intended by the dynamic registrations. This can result in chaining of user callbacks.

For this purpose, the `OCIUserCallbackGet()` function is provided to find out which function and context is registered for an OCI call.

See Also: "OCIUserCallbackGet()" on page 17-190

Accessing Other Data Sources Through OCI

Because Oracle is the predominant database accessed, applications can take advantage of the OCI interface to access non-Oracle data by using the user callbacks to access them. This allows an application written in OCI to access Oracle data without any performance penalty. To access non-Oracle data sources, drivers can be written that access the non-Oracle data in user callbacks. Because OCI provides a very rich interface, there is usually a straightforward mapping of OCI calls to most data sources. This solution is better than writing applications for other middle layers such as ODBC that introduce performance penalties for all data sources. Using OCI does not incur any penalty for the common case of accessing Oracle data sources, and incurs the same penalty that ODBC does for non-Oracle data sources.

Restrictions on Callback Functions

There are certain restrictions on the usage of callback functions, including `OCIEnvCallback()`:

- A callback cannot call other OCI functions except `OCIUserCallbackRegister()`, `OCIUserCallbackGet()`, `OCIHandleAlloc()`, `OCIHandleFree()`. Even for these functions, if they are called in a user callback, then callbacks on them are not called to avoid recursion. For example, if `OCIHandleFree()` is called in the callback for `OCILogoff()`, then the callback for `OCIHandleFree()` is disabled during the execution of the callback for `OCILogoff()`.
- A callback cannot modify OCI data structures such as the environment or error handles.
- A callback cannot be registered for `OCIUserCallbackRegister()` call itself, or for any of the following:
  - `OCIUserCallbackGet()`
  - `OCIEnvCreate()`
  - `OCIInitialize()`
  - `OCIEnvInit()`

Example of OCI Callbacks

For example, lets suppose that there are five packages each registering entry, replacement, and exit callbacks for the `OCIStmtPrepare()` call. That is, the `ORA_OCI_UCBPKG` variable is set as:

```
setenv ORA_OCI_UCBPKG "pkg1;pkg2;pkg3;pkg4;pkg5"
```

In each package `pkgN` (where N can be 1 through 5), the `pkgNInit()` and `PkgNEnvCallback()` functions are specified as:

```
pkgNInit(void *metaCtx, void *libCtx, ub4 argfmt, sword argc, void **argv)
{
  return OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv, pkgNEnvCallback);
}
```

The `pkgNEnvCallback()` function registers the entry, replacement, and exit callbacks as:

```
pkgNEnvCallback(OCIEnv *env, ub4 mode, size_t xtramemsz,
                                 void *usrmemp, OCIUcb *ucbDesc)
{
  OCIHandleAlloc((void *)env, (void **)&errh, OCI_HTYPE_ERROR, (size_t) 0,
        (void **)NULL);

  OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_entry_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_ENTRY, ucbDesc);

  OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_replace_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_REPLACE, ucbDesc);

  OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_exit_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_EXIT, ucbDesc);

  return OCI_CONTINUE;
}
```

Finally, in the source code for the application, user callbacks can be registered with the `NULL` *ucbDesc* as:

```
  OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_entry_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_ENTRY, (OCIUcb *)NULL);

  OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_replace_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_REPLACE, (OCIUcb *)NULL);

  OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_exit_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_EXIT, (OCIUcb *)NULL);
```

When the `OCIStmtPrepare()` call is executed, the callbacks are called in the following order:

```
static_entry_callback_fn()
pkg1_entry_callback_fn()
pkg2_entry_callback_fn()
pkg3_entry_callback_fn()
pkg4_entry_callback_fn()
pkg5_entry_callback_fn()

static_replace_callback_fn()
 pkg1_replace_callback_fn()
  pkg2_replace_callback_fn()
   pkg3_replace_callback_fn()
    pkg4_replace_callback_fn()
     pkg5_replace_callback_fn()

      OCI code for OCIStmtPrepare call

pkg5_exit_callback_fn()
```

```
pkg4_exit_callback_fn()
pkg3_exit_callback_fn()
pkg2_exit_callback_fn()
pkg1_exit_callback_fn()

static_exit_callback_fn()
```

Note: The exit callbacks are called in the reverse order of the entry and replacement callbacks

The entry and exit callbacks can return any return code and the processing continues to the next callback. However, if the replacement callback returns anything other than `OCI_CONTINUE`, then the next callback (or OCI code if it is the last replacement callback) in the chain is bypassed and processing jumps to the exit callback. For example, if `pkg3_replace_callback_fn()` returned `OCI_SUCCESS`, then `pkg4_replace_callback_fn()`, `pkg5_replace_callback_fn()`, and the OCI processing for the `OCIStmtPrepare()` call is bypassed. Instead, `pkg5_exit_callback_fn()` is executed next.

OCI Callbacks from External Procedures

There are several OCI functions that can be used as callbacks from external procedures.

See Also: These functions are listed in Chapter 20, "OCI Cartridge Functions". For information about writing C subroutines that can be called from PL/SQL code, including a list of which OCI calls can be used, and some example code, refer to the *Oracle Database Advanced Application Developer's Guide*

Transparent Application Failover in OCI

Transparent Application Failover (TAF) is a client-side feature designed to minimize disruptions to end-user applications that occur when database connectivity fails because of instance or network failure. TAF can be implemented on a variety of system configurations including Real Application Clusters and Data Guard Physical Standby Databases. TAF can also be used after re-starting a single instance system, for example when repairs are made.

TAF can be configured to restore database sessions and, optionally, replay open queries. In previous versions, TAF with the `SELECT` failover option would be engaged only on the statement that was in use at the time of a failure. For example, if there were 10 statement handles in use by the application, and statement 7 was the failure-time statement (the statement in use when the failure happened), statements 1-6 and 8-10 would have to be re-executed after statement 7 was failed over using TAF.

Starting with 10g Release 2 (10.2), this has been improved. Now all statements that an application attempts to use after a failure will attempt failover. That is, an attempt to execute or fetch against other statements will engage TAF recovery just as for the failure-time statement. Notably, this means that subsequent statements may now succeed (whereas in the past they failed), or the application may receive errors corresponding to an attempted TAF recovery (like ORA-25401).

Note: TAF is not supported for remote database links or for DML statements.

Configuring Transparent Application Failover

TAF can be configured on both the client-side and the server-side. If both are configured, server-side settings take precedence.

Configure TAF on the client side by including the `FAILOVER_MODE` parameter in the `CONNECT_DATA` portion of a connect descriptor.

> **See Also:** *Oracle Database Net Services Reference* for more information about client-side configuration of TAF (Connect Data Section).

Configure TAF on the server-side by modifying the target service with the `DBMS_SERVICE.MODIFY_SERVICE` packaged procedure.

An initial attempt at failover may not always be successful. The OCI provides a mechanism for retrying failover after an unsuccessful attempt.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information on the server-side configuration of TAF (DBMS_SERVICE).

Transparent Application Failover Callbacks in OCI

Because of the delay that can occur during failover, the application developer may want to inform the user that failover is in progress, and request that the user wait for notification that failover is complete. Additionally, the session on the initial instance may have received some `ALTER SESSION` commands. These `ALTER SESSION` commands will not be automatically replayed on the second instance. Consequently, the developer may wish to replay them on the second instance.

To accommodate these requirements, the application developer can register a failover callback function. In the event of failover, the callback function is invoked several times during the course of reestablishing the user's session.

The first call to the callback function occurs when the database first detects an instance connection loss. This callback is intended to allow the application to inform the user of an upcoming delay. If failover is successful, a second call to the callback function occurs when the connection is reestablished and usable.

Once the connection has been re-established, the client may wish to replay `ALTER SESSION` commands and inform the user that failover has happened. If failover is unsuccessful, then the callback is called to inform the application that failover will not take place. Additionally, the callback is called each time a user handle besides the primary handle is re-authenticated on the new connection. Since each user handle represents a server-side session, the client may wish to replay `ALTER SESSION` commands for that session.

> **See Also:** See "Handling OCI_FO_ERROR" on page 9-35 for more information about this scenario

Failover Callback Structure and Parameters

The basic structure of a user-defined application failover callback function is as follows:

```
sb4 appfocallback_fn ( void       * svchp,
                       void       * envhp,
                       void       * fo_ctx,
                       ub4        fo_type,
                       ub4        fo_event );
```

An example is provided in the section "Failover Callback Example" on page 9-34 for the following parameters:

svchp
The first parameter, `svchp`, is the service context handle. It is of type `void *`.

envhp
The second parameter, `envhp`, is the OCI environment handle. It is of type `void *`.

fo_ctx
The third parameter, `fo_ctx`, is a client context. It is a pointer to memory specified by the client. In this area the client can keep any necessary state or context. It is passed as a `void *`.

fo_type
The fourth parameter, `fo_type`, is the failover type. This lets the callback know what type of failover the client has requested. The usual values are:

- `OCI_FO_SESSION`, which indicates that the user has requested only session failover.
- `OCI_FO_SELECT`, which indicates that the user has requested select failover as well.

fo_event
The last parameter is the failover event. This indicates to the callback why it is being called. It has several possible values:

- `OCI_FO_BEGIN` indicates that failover has detected a lost connection and failover is starting.
- `OCI_FO_END` indicates successful completion of failover.
- `OCI_FO_ABORT` indicates that failover was unsuccessful, and there is no option of retrying.
- `OCI_FO_ERROR` also indicates that failover was unsuccessful, but it gives the application the opportunity to handle the error and retry failover.
- `OCI_FO_REAUTH` indicates that you have multiple authentication handles and failover has occurred after the original authentication. It indicates that a user handle has been re-authenticated. To find out which, the application checks the `OCI_ATTR_SESSION` attribute of the service context handle (which is the first parameter).

Failover Callback Registration

For the failover callback to be used, it must be registered on the server context handle. This registration is done by creating a callback definition structure and setting the `OCI_ATTR_FOCBK` attribute of the server handle to this structure.

The callback definition structure must be of type `OCIFocbkStruct`. It has two fields: `callback_function`, which contains the address of the function to call, and `fo_ctx` which contains the address of the client context.

An example of callback registration is included as part of the example in the next section.

Failover Callback Example

The following code shows an example of a simple user-defined callback function definition, registration, and unregistration.

Part 1: Failover Callback Definition

```
sb4  callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event)
void * svchp;
void * envhp;
void *fo_ctx;
ub4 fo_type;
ub4 fo_event;
{
switch (fo_event)
   {
   case OCI_FO_BEGIN:
   {
     printf(" Failing Over ... Please stand by \n");
     printf(" Failover type was found to be %s \n",
                    ((fo_type==OCI_FO_SESSION) ? "SESSION"
                    :(fo_type==OCI_FO_SELECT) ? "SELECT"
                    : "UNKNOWN!"));
     printf(" Failover Context is :%s\n",
                   (fo_ctx?(char *)fo_ctx:"NULL POINTER!"));
     break;
   }
   case OCI_FO_ABORT:
   {
     printf(" Failover stopped. Failover will not take place.\n");
     break;
   }
   case    OCI_FO_END:
   {
       printf(" Failover ended ...resuming services\n");
     break;
   }
   case OCI_FO_REAUTH:
   {
       printf(" Failed over user. Resuming services\n");
     break;
   }
   default:
   {
     printf("Bad Failover Event: %d.\n",  fo_event);
     break;
   }
   }
   return 0;
}
```

Part 2: Failover Callback Registration

```
int register_callback(srvh, errh)
void *srvh; /* the server handle */
OCIError *errh; /* the error handle */
{
  OCIFocbkStruct failover;                /*  failover callback structure */
  /* allocate memory for context */
  if (!(failover.fo_ctx = (void *)malloc(strlen("my context.")+1)))
     return(1);
```

```
  /* initialize the context. */
  strcpy((char *)failover.fo_ctx, "my context.");
  failover.callback_function = &callback_fn;
  /* do the registration */
  if (OCIAttrSet(srvh, (ub4) OCI_HTYPE_SERVER,
                 (void *) &failover, (ub4) 0,
                 (ub4) OCI_ATTR_FOCBK, errh)  != OCI_SUCCESS)
     return(2);
  /* successful conclusion */
  return (0);
}
```

Part 3: Failover Callback Unregistration

```
OCIFocbkStruct failover;   /*  failover callback structure */
sword status;

  /* set the failover context to null */
  failover.fo_ctx = NULL;
  /* set the failover callback to null */
  failover.callback_function = NULL;
  /* un-register the callback */
  status = OCIAttrSet(srvhp, (ub4) OCI_HTYPE_SERVER,
                      (void *) &failover, (ub4) 0,
                      (ub4) OCI_ATTR_FOCBK, errhp);
```

Handling OCI_FO_ERROR

A failover attempt is not always successful. If the attempt fails, the callback function receives a value of `OCI_FO_ABORT` or `OCI_FO_ERROR` in the `fo_event` parameter. A value of `OCI_FO_ABORT` indicates that failover was unsuccessful, and no further failover attempts are possible. `OCI_FO_ERROR`, on the other hand, provides the callback function with the opportunity to handle the error in some way. For example, the callback may choose to wait a specified period of time and then indicate to the OCI library that it must reattempt failover.

> **Note:** This functionality is only available to applications linked with the 8.0.5 or later OCI libraries running against any Oracle server.
>
> Failover does not work if a LOB column is part of the select list.

Consider the following time line of events:

Table 9–2 Time and Event

Time	Event
T0	Database fails (failure lasts until T5).
T1	Failover triggered by user activity.
T2	User attempts to reconnect; attempt fails.
T3	Failover callback invoked with `OCI_FO_ERROR`.
T4	Failover callback enters predetermined sleep period.
T5	Database comes back up again.
T6	Failover callback triggers new failover attempt; it is successful.

Table 9–2 (Cont.) Time and Event

Time	Event
T7	User successfully reconnects

The callback function triggers the new failover attempt by returning a value of OCI_FO_RETRY from the function.

The following example code shows a callback function which can be used to implement the failover strategy similar to the scenario described earlier. In this case the failover callback enters a loop in which it sleeps and then reattempts failover until it is successful:

```
/*-------------------------------------------------------------------------*/
/* the user defined failover callback  */
/*-------------------------------------------------------------------------*/
sb4  callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event )
void * svchp;
void * envhp;
void *fo_ctx;
ub4 fo_type;
ub4 fo_event;
{
   OCIError *errhp;
   OCIHandleAlloc(envhp, (void **)&errhp, (ub4) OCI_HTYPE_ERROR,
              (size_t) 0, (void **) 0);
   switch (fo_event)
   {
   case OCI_FO_BEGIN:
   {
     printf(" Failing Over ... Please stand by \n");
     printf(" Failover type was found to be %s \n",
            ((fo_type==OCI_FO_NONE) ? "NONE"
             :(fo_type==OCI_FO_SESSION) ? "SESSION"
             :(fo_type==OCI_FO_SELECT) ? "SELECT"
             :(fo_type==OCI_FO_TXNAL) ? "TRANSACTION"
             : "UNKNOWN!"));
     printf(" Failover Context is :%s\n",
            (fo_ctx?(char *)fo_ctx:"NULL POINTER!"));
     break;
   }
   case OCI_FO_ABORT:
   {
     printf(" Failover aborted. Failover will not take place.\n");
     break;
   }
   case    OCI_FO_END:
   {
       printf("\n Failover ended ...resuming services\n");
     break;
   }
   case OCI_FO_REAUTH:
   {
       printf(" Failed over user. Resuming services\n");
     break;
   }
   case OCI_FO_ERROR:
   {
     /* all invocations of this can only generate one line. The newline
      * will be put at fo_end time.
```

```
      */
      printf(" Failover error gotten. Sleeping...");
      sleep(3);
      printf("Retrying. ");
      return (OCI_FO_RETRY);
      break;
    }
    default:
    {
      printf("Bad Failover Event: %d.\n",  fo_event);
      break;
    }
    }
    return 0;
}
```

HA Event Notification

Suppose an user employs a Web browser to log in to an application server that accesses a back-end database server. Failure of the database instance can result in a wait that can be up to in minutes in duration before the failure is known to the user. The ability to quickly detect failures of server instances, communicate this to the client, close connections, and clean up idle connections in connection pools is provided by HA event notification.

For High Availability clients connected to a Real Application Clusters (RAC) database, HA event notification can be used to provide a best-effort programmatic signal to the client in the event of a database failure. Client applications can register a callback on the environment handle to signal interest in this information. When a significant failure event occurs (which applies to a connection made by this client), the callback is invoked, with information concerning the event (the event payload), and a list of connections (server handles) that were disconnected as a result of the failure.

For example, consider a client application that has two connections to instance A and two connections to instance B of the same database. If instance A goes down, a notification of the event will be sent to the client, which will then disconnect the two connections to instance B, and invoke the registered callback. Note that if another instance, C, of the same database, goes down, the client will not be notified (since it will not affect any of the client's connections).

The HA event notification mechanism improves the response time of the application in the presence of failure. In the past, a failure would result in the connection being broken only after the TCP time out expired, which could take minutes. With HA event notification, standalone, connection pool, and session pool connections are automatically broken and cleaned up by OCI and the application callback is invoked within seconds of the failure event. If any of these server handles are TAF-enabled, failover will also automatically be engaged by OCI.

Applications must connect to a RAC instance to enable HA event notification. Furthermore, these applications must:

- Initialize the OCI Environment in `OCI_EVENTS` mode.
- Connect to a service that has notifications enabled (use the `DBMS_SERVICE.MODIFY_SERVICE` procedure to set `AQ_HA_NOTIFICATIONS` to `TRUE`).
- Link with a thread library.

Then these applications can register a callback that is invoked whenever an HA event occurs.

OCIEvent Handle

The `OCIEvent` handle encapsulates the attributes from the event payload. OCI implicitly allocates this handle prior to calling the event callback, which can obtain the read-only attributes of the event by calling `OCIAttrGet()`. Memory associated with these attributes is only valid for the duration of the event callback.

See Also: "Event Handle Attributes" on page A-72

OCI Failover for Connection and Session Pools

A connection pool in an instance of Real Application Clusters (RAC) consists of a pool of connections connected to different instances of the RAC. Upon receiving the node failure notification, all the connections connected to that particular instance should be cleaned up. For the connections that are in use, OCI has to close the connections: transparent application failover (TAF) occurs immediately and those connections will be reestablished. The connections that are idle and in the free list of the pool have to be purged, so that a bad connection is never returned back to the user from the pool.

To accommodate custom connection pools, OCI will provide a callback function that can be registered on the environment handle. If registered, this callback is invoked when an HA event occurs. Sessions pools are treated the same way as connection pools. Note that server handles from OCI connection pools or session pools will not be passed to the callback. Hence in some cases, the callback could be called with an empty list of connections.

OCI Failover for Independent Connections

No special handling is required for independent connections; all such connections that are connected to failed instances are immediately disconnected. For idle connections, TAF will be engaged to reestablish the connection when the connection is used on a subsequent OCI call. Connections that are in use at the time of the failure event will be broken out immediately, so that TAF can begin. Note that this applies for the "in-use" connections of connection and session pools also.

Event Callback

The event callback, of type `OCIEventCallback`, has the following signature:

```
void evtcallback_fn (void     *evtctx,
                     OCIEvent  *eventhp );
```

where `evtctx` is the client context and `OCIEvent` is an event handle which is opaque to the OCI library. The other input argument is `eventhp`, the event handle: that is, the attributes associated with an event.

If registered, this function will be called once for each event. In the case of RAC HA events, this callback will be invoked after the affected connections have been disconnected. The following environment handle attributes are used to register an event callback and context, respectively:

- `OCI_ATTR_EVTCBK` is of datatype `OCIEventCallback *`. It is read-only.
- `OCI_ATTR_EVTCTX` is of datatype `void *`. It is also read-only.

```
text *myctx = "dummy context"; /* dummy context passed to callback fn */
```

```
...
/* OCI_ATTR_EVTCBK and OCI_ATTR_EVTCTX are read-only. */
OCIAttrSet(envhp, (ub4) OCI_HTYPE_ENV, (void *) evtcallback_fn,
           (ub4) 0, (ub4) OCI_ATTR_EVTCBK, errhp);
OCIAttrSet(envhp, (ub4) OCI_HTYPE_ENV, (void *) myctx,
           (ub4) 0, (ub4) OCI_ATTR_EVTCTX, errhp);
...
```

Within the OCI Event callback, the list of affected server handles is encapsulated in the `OCIEvent` handle. For RAC HA DOWN events, client applications can iterate over a list of server handles that are affected by the event by using `OCIAttrGet()` with attribute types `OCI_ATTR_HA_SVRFIRST` and `OCI_ATTR_HA_SVRNEXT`:

```
OCIAttrGet(eventhp, OCI_HTYPE_EVENT, (void *)&srvhp, (ub4 *)0,
           OCI_ATTR_HA_SRVFIRST, errhp);
/* or, */
OCIAttrGet(eventhp, OCI_HTYPE_EVENT, (void *)&srvhp, (ub4 *)0,
           OCI_ATTR_HA_SRVNEXT, errhp);
```

When called with attribute `OCI_ATTR_HA_SRVFIRST`, this function will retrieve the first server handle in the list of server handles affected. When called with attribute `OCI_ATTR_HA_SRVNEXT`, this function will retrieve the next server handle in the list. This function will return `OCI_NO_DATA` and `srvhp` will be a `NULL` pointer, when there are no more server handles to return.

`srvhp` is an output pointer to a server handle whose connection has been closed as a result of an HA event. `errhp` is an error handle to populate. The application returns an `OCI_NO_DATA` error when there are no more affected server handles to retrieve.

When retrieving the list of server handles that have been affected by an HA event, be aware that the connection has already been closed and many server handle attributes are no longer valid. Instead, use the user memory segment of the server handle to store any per-connection attributes required by the event notification callback. This memory remains valid until the server handle is freed.

Custom Pooling: Tagged Server Handles

The following features apply to custom pools:

1. You can tag a server handle with its parent connection object if it is created on behalf of a custom pool. Use the "user memory" parameters of `OCIHandleAlloc()` to request that the server handle be allocated with a user memory segment. A pointer to the "user memory" segment is returned by `OCIHandleAlloc()`.
2. When an HA event occurs, and an affected server handle has been retrieved, there is a means to retrieve the server handle's tag information so appropriate cleanup can be performed. The attribute `OCI_ATTR_USER_MEMORY` is used to retrieve a pointer to a handle's user memory segment. `OCI_ATTR_USER_MEMORY` is valid for all user-allocated handles. If the handle was allocated with extra memory, this attribute will return a pointer to the user memory. A `NULL` pointer will be returned for those handles not allocated with extra memory. This attribute is read-only and is of datatype `void*`.

Note: You are free to define the precise contents of the server handle's user memory segment to facilitate cleanup activities from within the HA event callback (or for other purposes if needed) because OCI does not write or read from this memory in any way. The user memory segment is freed along with the `OCIHandleFree()` call on the server handle.

Event Notification Example

```
sword retval;
OCIServer *srvhp;
struct myctx {
   void *parentConn_myctx;
   uword numval_myctx;
};
typedef struct myctx myctx;
myctx  *myctxp;
/* Allocate a server handle with user memory - pre 10.2 functionality */
if (retval = OCIHandleAlloc(envhp, (void **)&srvhp, OCI_HTYPE_SERVER,
                            (size_t)sizeof(myctx), (void **)&myctxp)
/* handle error */
myctxp->parentConn_myctx = <parent connection reference>;

/* In an event callback function, retrieve the pointer to the user memory */
evtcallback_fn(void *evtctx, OCIEvent *eventhp)
{
  myctx *ctxp = (myctx *)evtctx;
  OCIServer *srvhp;
  OCIError *errhp;
  sb4       retcode;
  retcode = OCIAttrGet(eventhp, OCI_HTYPE_SERVER, &srvhp, (ub4 *)0,
                       OCI_ATTR_HA_SRVFIRST, errhp);
  while (!retcode) /* OCIAttrGet will return OCI_NO_DATA if no more srvhp */
  {
     OCIAttrGet((void *)srvhp, OCI_HTYPE_SERVER, (void *)&ctxp,
                (ub4)0, (ub4)OCI_ATTR_USER_MEMORY, errhp);
           /* Remove the server handle from the parent connection object */
     retcode = OCIAttrGet(eventhp, OCI_HTYPE_SERVER, &srvhp, (ub4 *)0,
                          OCI_ATTR_HA_SRVNEXT, errhp);
...
  }
...
}
```

Determining Transparent Application Failover (TAF) Capabilities

You can have the application adjust its behavior if a connection is or is not TAF-enabled. Use `OCIAttrGet()` as follows to find out if a server handle is TAF-enabled:

```
boolean taf_capable;
...
OCIAttrGet(srvhp, (ub4) OCI_HTYPE_SERVER, (void *) &taf_capable,
           (ub4) sizeof(taf_capable), (ub4)OCI_ATTR_TAF_ENABLED, errhp);
...
```

where `taf_capable` is a boolean variable, which this call sets to `TRUE` if the server handle is TAF-enabled, and `FALSE` if not; `srvhp` is an input target server handle;

OCI_ATTR_TAF_ENABLED is a an attribute which is a pointer to a boolean variable and is read-only; errhp is an input error handle.

OCI and Streams Advanced Queuing

The OCI provides an interface to Streams Advanced Queuing (Streams AQ) feature. Streams AQ provides message queuing as an integrated part of the Oracle server. Streams AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database*. By providing an integrated solution Streams AQ frees application developers to devote their efforts to their specific business logic rather than having to construct a messaging infrastructure.

Note: In order to use Streams Advanced Queuing, you must be using the Enterprise Edition

---

See Also:

- *Oracle Streams Advanced Queuing User's Guide*
- *Oracle XML Developer's Kit Programmer's Guide*
- For example code demonstrating the use of the OCI with AQ, refer to the description of OCIAQEnq() on page 17-106

OCI Streams Advanced Queuing Functions

The OCI library includes several functions related to Streams Advanced Queuing:

- OCIAQEnq()
- OCIAQDeq()
- OCIAQListen()
- OCIAQListen2()
- OCIAQEnqArray()
- OCIAQDeqArray()

You can enqueue an array of messages to a single queue. The messages all share the same enqueue options, but each message in the array can have different message properties. You can also dequeue an array of messages from a single queue. For transaction group queues, you can dequeue all messages for a single transaction group using one call.

See Also: "Streams Advanced Queuing and Publish-Subscribe Functions" on page 17-101

OCI Streams Advanced Queuing Descriptors

The following descriptors are used by OCI Streams AQ operations:

- OCIAQEnqOptions
- OCIAQDeqOptions
- OCIAQMsgProperties

- OCIAQAgent

You can allocate these descriptors with respect to the service handle using the standard OCIDescriptorAlloc() call. The following code shows examples of this:

```
OCIDescriptorAlloc(svch, &enqueue_options, OCI_DTYPE_AQENQ_OPTIONS, 0, 0 );
OCIDescriptorAlloc(svch, &dequeue_options, OCI_DTYPE_AQDEQ_OPTIONS, 0, 0 );
OCIDescriptorAlloc(svch, &message_properties, OCI_DTYPE_AQMSG_PROPERTIES, 0, 0);
OCIDescriptorAlloc(svch, &agent, OCI_DTYPE_AQAGENT, 0, 0 );
```

Each descriptor has a variety of attributes which can be set or read.

> **See Also:** "Streams Advanced Queuing Descriptor Attributes" on page A-38

Streams Advanced Queuing in OCI Versus PL/SQL

The following tables compare functions, parameters, and options for OCI Streams AQ functions and descriptors, and PL/SQL AQ functions in the DBMS_AQ package.

Table 9–3 AQ Functions

PL/SQL Function	OCI Function
DBMS_AQ.ENQUEUE	OCIAQEnq()
DBMS_AQ.DEQUEUE	OCIAQDeq()
DBMS_AQ.LISTEN	OCIAQListen(), OCIAQListen2()
DBMS_AQ.ENQUEUE_ARRAY	OCIAQEnqArray()
DBMS_AQ.DEQUEUE_ARRAY	OCIAQDeqArray()

The following table compares the parameters for the enqueue functions:

Table 9–4 Enqueue Parameters

DBMS_AQ.ENQUEUE Parameter	OCIAQEnq() Parameter
queue_name	queue_name
enqueue_options	enqueue_options
message_properties	message_properties
payload	payload
msgid	msgid
-	Note: OCIAQEnq() also requires the following additional parameters: svch, errh, payload_tdo, payload_ind, and flags.

The following table compares the parameters for the dequeue functions:

Table 9–5 Dequeue Parameters

DBMS_AQ.DEQUEUE Parameter	OCIAQDeq() Parameter
queue_name	queue_name
dequeue_options	dequeue_options
message_properties	message_properties

Table 9–5 (Cont.) Dequeue Parameters

DBMS_AQ.DEQUEUE Parameter	OCIAQDeq() Parameter
payload	payload
msgid	msgid
-	Note: OCIAQDeq() also requires the following additional parameters: svch, errh, dequeue_options, message_properties, payload_tdo, payload, payload_ind, and flags.

The following table compares parameters for the listen functions:

Table 9–6 Listen Parameters

DBMS_AQ.LISTEN Parameter	OCIAQListen2() Parameter
agent_list	agent_list
wait	wait
agent	agent
listen_delivery_mode	lopts
-	Note: OCIAQListen2() also requires the following additional parameters: svchp, errhp, agent_list, num_agents, agent, lmops, and flags.

The following table compares parameters for the array enqueue functions:

Table 9–7 Array Enqueue Parameters

DBMS_AQ.ENQUEUE_ARRAY Parameter	OCIAQEnqArray() Parameter
queue_name	queue_name
enqueue_options	enqopt
array_size	iters
message_properties_array	msgprop
payload_array	payload
msgid_array	msgid
-	Note: OCIAQEnqArray() also requires the following additional parameters: svch, errh, payload_tdo, payload_ind, ctxp, enqcbfp, and flags.

The following table compares parameters for the array dequeue functions:

Table 9–8 Array Dequeue Parameters

DBMS_AQ.DEQUEUE_ARRAY Parameter	OCIAQDeqArray() Parameter
queue_name	queue_name
dequeue_options	deqopt

Table 9–8 (Cont.) Array Dequeue Parameters

DBMS_AQ.DEQUEUE_ARRAY Parameter	OCIAQDeqArray() Parameter
array_size	iters
message_properties_array	msgprop
payload_array	payload
msgid_array	msgid
-	Note: OCIAQDeqArray() also requires the following additional parameters: svch, errh, msgprop, payload_tdo, payload_ind, ctxp, deqcbfp, and flags.

The following table compares parameters for the agent attributes:

Table 9–9 Agent Parameters

PL/SQL Agent Parameter	OCIAQAgent Attribute
name	OCI_ATTR_AGENT_NAME
address	OCI_ATTR_AGENT_ADDRESS
protocol	OCI_ATTR_AGENT_PROTOCOL

The following table compares parameters for the message properties:

Table 9–10 Message Properties

PL/SQL Message Property	OCIAQMsgProperties Attribute
priority	OCI_ATTR_PRIORITY
delay	OCI_ATTR_DELAY
expiration	OCI_ATTR_EXPIRATION
correlation	OCI_ATTR_CORRELATION
attempts	OCI_ATTR_ATTEMPTS
recipient_list	OCI_ATTR_RECIPIENT_LIST
exception_queue	OCI_ATTR_EXCEPTION_QUEUE
enqueue_time	OCI_ATTR_ENQ_TIME
state	OCI_ATTR_MSG_STATE
sender_id	OCI_ATTR_SENDER_ID
transaction_group	OCI_ATTR_TRANSACTION_NO
original_msgid	OCI_ATTR_ORIGINAL_MSGID
delivery_mode	OCI_ATTR_MSG_DELIVERY_MODE

The following table compares enqueue option attributes:

Table 9–11 Enqueue Option Attributes

PL/SQL Enqueue Option	OCIAQEnqOptions Attribute
visibility	OCI_ATTR_VISIBILITY

Table 9–11 (Cont.) Enqueue Option Attributes

PL/SQL Enqueue Option	OCIAQEnqOptions Attribute
relative_msgid	OCI_ATTR_RELATIVE_MSGID
sequence_deviation	OCI_ATTR_SEQUENCE_DEVIATION (deprecated)

The following table compares dequeue option attributes:

Table 9–12 Dequeue Option Attributes

PL/SQL Dequeue Option	OCIAQDeqOptions Attribute
consumer_name	OCI_ATTR_CONSUMER_NAME
dequeue_mode	OCI_ATTR_DEQ_MODE
navigation	OCI_ATTR_NAVIGATION
visibility	OCI_ATTR_VISIBILITY
wait	OCI_ATTR_WAIT
msgid	OCI_ATTR_DEQ_MSGID
correlation	OCI_ATTR_CORRELATION
delivery_mode	OCI_ATTR_MSG_DELIVERY_MODE

Note: OCIAQEnq() returns the error ORA-25219 while specifying the enqueue option OCI_ATTR_SEQUENCE along with OCI_ATTR_RELATIVE_MSGID. This happens when enqueuing two messages. For the second message, enqueue options OCI_ATTR_SEQUENCE and OCI_ATTR_RELATIVE_MSGID is set to dequeue this message before the first one. It does not return an error if you do not specify the 'sequence' but, of course, the message is not dequeued before the relative message.

OCIAQEnq() does not return an error if OCI_ATTR_SEQUENCE attribute is not set, but the message is not dequeued before the message with relative message Id.

Buffered Messaging

Buffered messaging is a non-persistent messaging capability within Streams AQ, that was first available in Oracle Database 10*g* Release 2. Buffered messages reside in shared memory, and can be lost in the event of instance failure. Unlike persistent messages, no redo is written to disk. Buffered message enqueue and dequeue is much faster than persistent message operations. Since shared memory is limited, buffered messages may have to be spilled to disk. Flow control may be enabled to prevent applications from flooding the shared memory when the message consumers are slow or have stopped for some reason. The following functions are used for buffered messaging:

See Also:

- "OCIAQEnq()" on page 17-106
- "OCIAQDeq()" on page 17-102
- "OCIAQListen2()" on page 17-111

Enqueue Buffered Messaging Example

```
...
OCIAQMsgProperties  *msgprop;
OCIAQEnqueueOptions *enqopt;
message              msg;    /* message is an object type */
null_message         nmsg;   /* message indicator */
...
/* Allocate descriptors */
  OCIDescriptorAlloc(envhp, (void **)&enqopt, OCI_DTYPE_AQENQ_OPTIONS, 0,
                     (void **)0));

 OCIDescriptorAlloc(envhp, (void **)&msgprop,OCI_DTYPE_AQMSG_PROPERTIES, 0,
                    (void **)0));

/* Set delivery mode to buffered */
 dlvm = OCI_MSG_BUFFERED;
 OCIAttrSet(enqopt,  OCI_DTYPE_AQENQ_OPTIONS, (void *)&dlvm, sizeof(ub2),
            OCI_ATTR_MSG_DELIVERY_MODE, errhp);
/* Set visibility to Immediate (visibility must always be immediate for buffered
   messages) */
vis = OCI_ENQ_ON_COMMIT;

OCIAttrSet(enqopt, OCI_DTYPE_AQENQ_OPTIONS,(void *)&vis, sizeof(ub4),
           OCI_ATTR_VISIBILITY, errhp)

/* Message was an object type created earlier, msg_tdo is its Type
   Descriptor Object */
OCIAQEnq(svchp, errhp, "Test_Queue", enqopt, msgprop, msg_tdo, (void **)&mesg,
         (void **)&nmesg, (OCIRaw **)0, 0));
...
```

Dequeue Buffered Messaging Example

```
...
OCIAQMsgProperties  *msgprop;
OCIAQDequeueOptions *deqopt;
...
OCIDescriptorAlloc(envhp, (void **)&mprop, OCI_DTYPE_AQMSG_PROPERTIES, 0,
                   (void **)0));
OCIDescriptorAlloc(envhp, (void **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
                   (void **)0);

/* Set visibility to Immediate (visibility must always be immediate for buffered
   message operations) */
vis = OCI_ENQ_ON_COMMIT;
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,(void *)&vis, sizeof(ub4),
           OCI_ATTR_VISIBILITY, errhp)
/* delivery mode is buffered */
dlvm  = OCI_MSG_BUFFERED;
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (void *)&dlvm,  sizeof(ub2),
           OCI_ATTR_MSG_DELIVERY_MODE, errhp);
/* set the consumer for which to dequeue the message (this needs to be specified
   regardless of the type of message being dequeued.
```

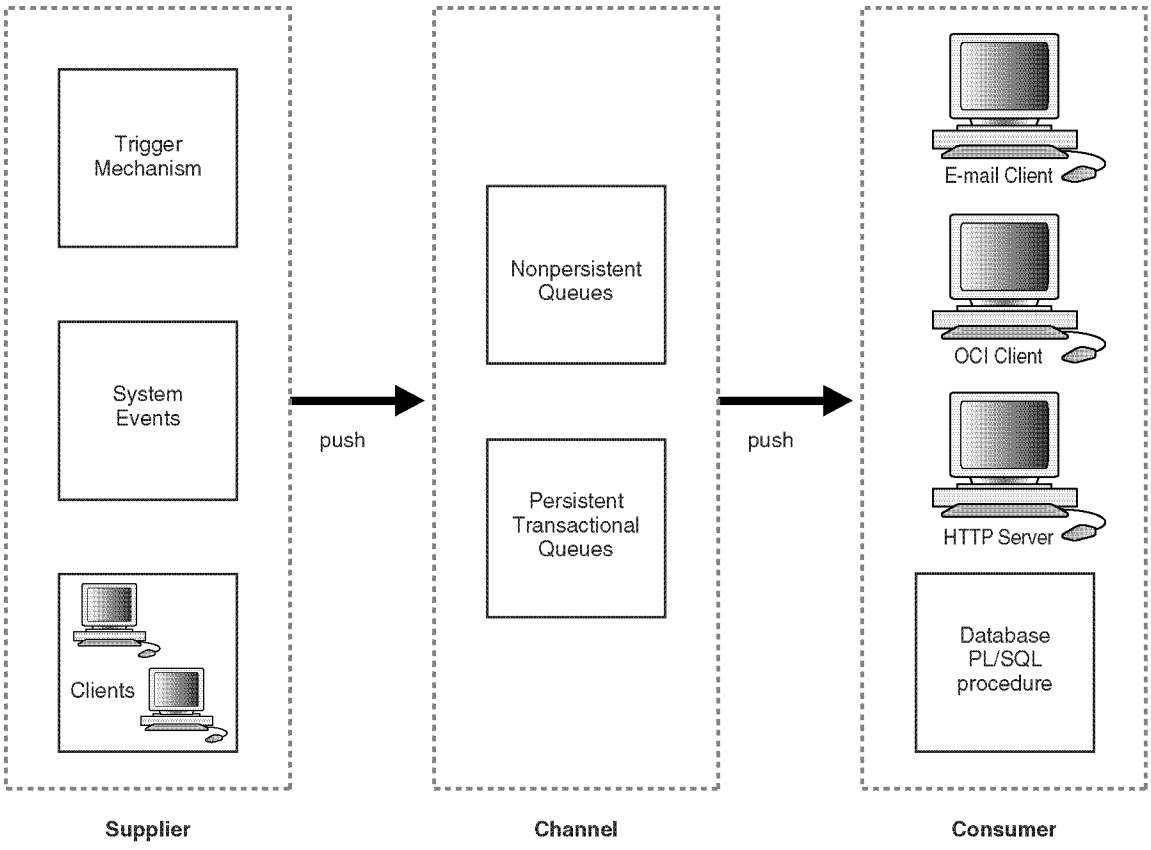
```
*/
consumer = "FIRST_SUBSCRIBER";
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (void *)consumer,
           (ub4)strlen((char*)consumer), OCI_ATTR_CONSUMER_NAME, errhp);
/* Dequeue the message but don't return the payload (to simplify the code
   snippet)
*/
OCIAQDeq(svchp, errhp,  "test_queue", deqopt, msgprop, msg_tdo, (void **)0,
         (void **)0, (OCIRaw**)0, 0);
...
```

Note: Array operations are not supported for buffered messaging. Applications can use the `OCIAQEnqArray()` and `OCIAQDeqArray()` functions with array size set to 1.

Publish-Subscribe Notification in OCI

The publish-subscribe notification feature allows an OCI application to receive client notifications directly, register an e-mail address to which notifications can be sent, register a HTTP URL to which notifications can be posted, or register a PL/SQL procedure to be invoked on a notification. Figure 9–2, "Publish-Subscribe Model" illustrates the process.

Figure 9–2 Publish-Subscribe Model



An OCI application can:

- register interest in notifications in the AQ namespace and be notified when an enqueue occurs.
- register interest in subscriptions to database events and receive notifications when the events are triggered.
- manage registrations, such as disabling registrations temporarily or dropping the registrations entirely.
- post, or send, notifications to registered clients.

In all the preceding scenarios the notification can be received directly by the OCI application, or the notification can be sent to a pre-specified e-mail address, or it can be sent to a pre-defined HTTP URL, or a pre-specified database PL/SQL procedure can be invoked as a result of a notification.

Registered clients are notified asynchronously when events are triggered or on an explicit AQ enqueue. Clients do not need to be connected to a database.

See Also:

- For information on Streams Advanced Queuing, see "OCI and Streams Advanced Queuing" on page 9-41
- For information on creating queues and about Streams AQ, including concepts, features, and examples, refer to *Oracle Streams Advanced Queuing User's Guide*
- For information on creating triggers, refer to the chapter on `CREATE TRIGGER` in the *Oracle Database SQL Language Reference*.

Publish-Subscribe Registration Functions in OCI

Registration can be done in two ways:

- You register directly to the database. This way is simple and the registration will take effect immediately.
- Open Registration. You register using LDAP, from which the database receives the registration request. This is useful when the client cannot have a database connection (the client wants to register for a database open event while the database is down), or if the client wants to register for the same event or events in multiple databases at one time.

Let us next consider these two alternative ways of registration.

Publish-Subscribe Register Directly to the Database

The following steps are required in an OCI application to register and receive notifications for events. It is assumed that the appropriate event trigger or AQ queue has been set up. The initialization parameter `COMPATIBLE` must be set to 8.1 or higher.

See Also:

- Detailed descriptions of the functions noted can be found in "Streams Advanced Queuing and Publish-Subscribe Functions" on page 17-101
- For examples of the use of these functions in an application, see "Publish-Subscribe Direct Registration Example" on page 9-57

Note: The publish-subscribe feature is only available on multithreaded operating systems.

1. Call `OCIInitialize()` with `OCI_EVENTS` mode to specify that the application is interested in registering for and receiving notifications. This starts a dedicated listening thread for notifications on the client.
2. Call `OCIHandleAlloc()` with handle type `OCI_HTYPE_SUBSCRIPTION` to allocate a subscription handle.
3. Call `OCIAttrSet()` to set the subscription handle attributes for:
   - `OCI_ATTR_SUBSCR_NAME` - subscription name
   - `OCI_ATTR_SUBSCR_NAMESPACE` - subscription namespace
   - `OCI_ATTR_SUBSCR_CALLBACK` - notification callback
   - `OCI_ATTR_SUBSCR_CTX` - callback context
   - `OCI_ATTR_SUBSCR_PAYLOAD` - payload buffer for posting
   - `OCI_ATTR_SUBSCR_RECPT` - recipient name
   - `OCI_ATTR_SUBSCR_RECPTPROTO` - protocol to receive notification with
   - `OCI_ATTR_SUBSCR_RECPTPRES` - presentation to receive notification with
   - `OCI_ATTR_SUBSCR_QOSFLAGS` - QOS (quality of service) levels
   - `OCI_ATTR_SUBSCR_TIMEOUT` - Registration timeout interval in seconds. The default is 0 if a timeout is not set.
   - `OCI_ATTR_SUBSCR_NTFN_GROUPING_CLASS` - notification grouping class

     Notifications can be spaced out by using the grouping NTFN option with the following constants. A value supported for notification grouping class is:

     ```
     #define OCI_SUBSCR_NTFN_GROUPING_CLASS_TIME   1 /* time  */
     ```

   - `OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE` - notification grouping value in seconds
   - `OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE` - notification grouping type

     Supported values for notification grouping type:

     ```
     #define OCI_SUBSCR_NTFN_GROUPING_TYPE_SUMMARY 1  /* summary */
     #define OCI_SUBSCR_NTFN_GROUPING_TYPE_LAST    2  /* last */
     ```

   - `OCI_ATTR_SUBSCR_NTFN_GROUPING_START_TIME` - notification grouping start time
   - `OCI_ATTR_SUBSCR_NTFN_GROUPING_REPEAT_COUNT` - notification grouping repeat count
   - If `OCI_SUBSCR_QOS_PURGE_ON_NTFN` is set, the registration is purged on the first notification.
   - If `OCI_SUBSCR_QOS_RELIABLE` is set, notifications are persistent. Surviving instances of a RAC can be used to send and retrieve change notification messages, even after a node failure because invalidations associated with this registration are queued persistently into the database. If `FALSE`, then invalidations are enqueued into a fast in-memory queue. Note that this option

describes the persistence of notifications and not the persistence of registrations. Registrations are automatically persistent by default.

`OCI_ATTR_SUBSCR_NAME`, `OCI_ATTR_SUBSCR_NAMESPACE` and `OCI_ATTR_SUBSCR_RECPTPROTO` must be set before registering a subscription.

If `OCI_ATTR_SUBSCR_RECPTPROTO` is set to `OCI_SUBSCR_PROTO_OCI`, then `OCI_ATTR_SUBSCR_CALLBACK` and `OCI_ATTR_SUBSCR_CTX` also need to be set.

If `OCI_ATTR_SUBSCR_RECPTPROTO` is set to `OCI_SUBSCR_PROTO_MAIL`, `OCI_SUBSCR_PROTO_SERVER`, or `OCI_SUBSCR_PROTO_HTTP`, then `OCI_ATTR_SUBSCR_RECPT` also needs to be set.

Setting `OCI_ATTR_SUBSCR_CALLBACK` and `OCI_ATTR_SUBSCR_RECPT` at the same time will cause an application error.

`OCI_ATTR_SUBSCR_PAYLOAD` is required before posting to a subscription.

See Also: "Subscription Handle Attributes" on page A-48 and "Creating the OCI Environment" on page 2-15 for setting up the environment with `mode = OCI_EVENTS | OCI_OBJECT`. `OCI_OBJECT` is required for grouping notifications.

4. The values of QOS, timeout interval, namespace, and port are set:

   See Also: "Setting QOS, Timeout Interval, Namespace, and Port Number" on page 9-53

5. If `OCI_ATTR_SUBSCR_RECPTPROTO` is set to `OCI_SUBSCR_PROTO_OCI`, then define the callback routine to be used with the subscription handle.

   See Also: "Notification Callback in OCI" on page 9-54

6. If `OCI_ATTR_SUBSCR_RECPTPROTO` is set to `OCI_SUBSCR_PROTO_SERVER`, then define the PL/SQL procedure, to be invoked on notification, in the database.

   See Also: "Notification Procedure" on page 9-56

7. Call `OCISubscriptionRegister()` to register with the subscriptions. This call can register interest in several subscriptions at the same time.

Example of Setting QOS Levels

```
/* Set QOS levels */
ub4 qosflags = OCI_SUBSCR_QOS_PAYLOAD;

/* Set QOS flags in subscription handle */
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *) &qosflags, (ub4) 0,
                  (ub4) OCI_ATTR_SUBSCR_QOSFLAGS, errhp);

/* Set notification grouping class */
ub4 ntfn_grouping_class = OCI_SUBSCR_NTFN_GROUPING_CLASS_TIME;
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *) &ntfn_grouping_class, (ub4) 0,
                  (ub4) OCI_ATTR_SUBSCR_NTFN_GROUPING_CLASS, errhp);

/* Set notification grouping value of 10 minutes */
ub4 ntfn_grouping_value = 600;
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
```

```
                (dvoid *) &ntfn_grouping_value, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NTFN_GROUPING_VALUE, errhp);

/* Set notification grouping type */
ub4 ntfn_grouping_type = OCI_SUBSCR_NTFN_GROUPING_TYPE_SUMMARY;

/* Set notification grouping type in subscription handle */
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &ntfn_grouping_type, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NTFN_GROUPING_TYPE, errhp);

/* Set namespace specific context */
(void) OCIAttrSet((dvoid *) subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) NULL, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NAMESPACE_CTX, errhp);
```

Open Registration for Publish-Subscribe

Prerequisites for this method are:

- Registering using LDAP (open registration) requires the client to be an enterprise user.

  See Also: *Oracle Database Advanced Security Administrator's Guide*, sections on managing enterprise user security

- The compatibility of the database has to be 9.0 or higher.
- LDAP_REGISTRATION_ENABLED must be set to TRUE. This can be done this way:

  ```
  ALTER SYSTEM SET LDAP_REGISTRATION_ENABLED=TRUE
  ```

  The default is `FALSE`.

- `LDAP_REG_SYNC_INTERVAL` must be set to the time interval (in seconds) to refresh registrations from LDAP:

  ```
  ALTER SYSTEM SET LDAP_REG_SYNC_INTERVAL =  time_interval
  ```

  The default is 0, which means do not refresh.

- To force a database refresh of LDAP registration information immediately:

  ```
  ALTER SYSTEM REFRESH LDAP_REGISTRATION
  ```

The steps in open registration using Oracle Enterprise Security Manager (OESM) are:

1. In each enterprise domain, create enterprise role, ENTERPRISE_AQ_USER_ROLE.
2. For each database in the enterprise domain, add global role GLOBAL_AQ_USER_ROLE to enterprise role ENTERPRISE_AQ_USER_ROLE.
3. For each enterprise domain, add enterprise role ENTERPRISE_AQ_USER_ROLE to privilege group cn=OracleDBAQUsers, under cn=oraclecontext, under the administrative context.
4. For each enterprise user that is authorized to register for events in the database, grant enterprise role ENTERPRISE_AQ_USER_ROLE.

Using OCI to Open Register with LDAP

1. Call `OCIInitialize()` with `mode` set to `OCI_EVENTS | OCI_USE_LDAP`.

2. Call `OCIAttrSet()` to set the following environment handle attributes for accessing LDAP:
   - `OCI_ATTR_LDAP_HOST`: the host name on which the LDAP server resides
   - `OCI_ATTR_LDAP_PORT`: the port on which the LDAP server is listening
   - `OCI_ATTR_BIND_DN`: the distinguished name to login to the LDAP server, usually the DN of the enterprise user
   - `OCI_ATTR_LDAP_CRED`: the credential used to authenticate the client, for example, the password for simple authentication (user name and password)
   - `OCI_ATTR_WALL_LOC`: for SSL authentication, the location of the client wallet
   - `OCI_ATTR_LDAP_AUTH`: the authentication method code

     See Also: "Environment Handle Attributes" on page A-2 for a complete list of authentication modes

   - `OCI_ATTR_LDAP_CTX`: the administrative context for Oracle in the LDAP server
3. Call `OCIHandleAlloc()` with handle type `OCI_HTYPE_SUBSCRIPTION`, to allocate a subscription handle.
4. Call `OCIDescriptorAlloc()` with descriptor type `OCI_DTYPE_SRVDN`, to allocate a server DN descriptor.
5. Call `OCIAttrSet()` to set the server DN descriptor attributes for `OCI_ATTR_SERVER_DN`, the distinguished name of the database in which the client wants to receive notifications. `OCIAttrSet()` can be called multiple times for this attribute so that more than one database server is included in the registration
6. Call `OCIAttrSet()` to set the subscription handle attributes for:
   - `OCI_ATTR_SUBSCR_NAME` - subscription name
   - `OCI_ATTR_SUBSCR_NAMESPACE` - subscription namespace
   - `OCI_ATTR_SUBSCR_CALLBACK` - notification callback
   - `OCI_ATTR_SUBSCR_CTX` - callback context
   - `OCI_ATTR_SUBSCR_PAYLOAD` - payload buffer for posting
   - `OCI_ATTR_SUBSCR_RECPT` - recipient name
   - `OCI_ATTR_SUBSCR_RECPTPROTO` - protocol to receive notification
   - `OCI_ATTR_SUBSCR_RECPTRES` - presentation to receive notification with
   - `OCI_ATTR_SUBSCR_QOSFLAGS` - QOS (quality of service) levels
   - `OCI_ATTR_SUBSCR_TIMEOUT` - Registration timeout interval in seconds. The default is 0 if a timeout is not set.
   - `OCI_ATTR_SUBSCR_SERVER_DN` - the descriptor handles you populated in step 5
7. The values of QOS, timeout interval, namespace, and port are set:

   See Also: "Setting QOS, Timeout Interval, Namespace, and Port Number" on page 9-53

8. Call `OCISubscriptionRegister()` to register the subscriptions. The registration will take effect when the database accesses LDAP to pick up new registrations. The frequency of pick-ups is determined by the value of REG_SYNC_INTERVAL.

Setting QOS, Timeout Interval, Namespace, and Port Number

You can set QOSFLAGS to the following QOS levels using `OCIAttrSet()`:

- `OCI_SUBSCR_QOS_RELIABLE` - Reliable notification persists across instance and database restarts. Reliability is of the server only and is only for persistent queues or buffered messages. This option describes the persistence of the notifications. Registrations are persistent by default.
- `OCI_SUBSCR_QOS_PURGE_ON_NTFN` - Once received, purge registration on first notification. (Subscription is unregistered.)

```
/* Set QOS levels */
ub4 qosflags = OCI_SUBSCR_QOS_RELIABLE | OCI_SUBSCR_QOS_PURGE_ON_NTFN;

/* Set flags in subscription handle */
(void)OCIAttrSet((void *)subscrhp, (ub4)OCI_HTYPE_SUBSCRIPTION,
                 (void *)&qosflags, (ub4)0, (ub4)OCI_ATTR_SUBSCR_QOSFLAGS, errhp);

/* Set auto-expiration after 30 seconds */
ub4 timeout = 30;
(void)OCIAttrSet((void *)subscrhp, (ub4)OCI_HTYPE_SUBSCRIPTION,
                  (void *)&timeout, (ub4)0, (ub4)OCI_ATTR_SUBSCR_TIMEOUT, errhp);
```

The registration is purged when the timeout is exceeded, and a notification is sent to the client, so that the client can invoke its callback and take any necessary action. In the case of client failure before the timeout, the registration is purged.

You can set the port number on the environment handle, which is important if the client is on a system behind a firewall that is able to receive notifications only on certain ports. Clients can specify the port for the listener thread before the first registration, using an attribute in the environment handle. The thread is started the first time `OCISubscriptionRegister()` is called. If available, this specified port number will be used. An error is returned if the client tries to start another thread on a different port using a different environment handle.

```
ub4 port = 1581;
(void)OCIAttrSet((void *)envhp, (ub4)OCI_HTYPE_ENV, (void *)&port, (ub4)0,
                  (ub4)OCI_ATTR_SUBSCR_PORTNO, errhp);
```

If instead, the port is determined automatically, you can get the port number at which the client thread is listening for notification by obtaining the attribute from the environment handle.

```
(void)OCIAttrGet((void *)subhp, (ub4)OCI_HTYPE_ENV, (void *)&port, (ub4)0,
                  (ub4)OCI_ATTR_SUBSCR_PORTNO, errhp);
```

OCI Functions Used to Manage Publish-Subscribe Notification

The following functions are used to manage publish-subscribe notification.

Table 9–13 Publish-Subscribe Functions

Function	Purpose
OCISubscriptionDisable()	Disables a subscription.

Table 9–13 (Cont.) Publish-Subscribe Functions

Function	Purpose
OCISubscriptionEnable()	Enables a subscription.
OCISubscriptionPost()	Posts a subscription.
OCISubscriptionRegister()	Registers a subscription.
OCISubscriptionUnRegister()	Unregisters a subscription.

Notification Callback in OCI

The client needs to register a notification callback that gets invoked when there is some activity on the subscription for which interest has been registered. In the AQ namespace, for instance, this occurs when a message of interest is enqueued.

This callback is typically set through the `OCI_ATTR_SUBSCR_CALLBACK` attribute of the subscription handle.

See Also: "Subscription Handle Attributes" on page A-48

The callback must return a value of `OCI_CONTINUE` and adhere to the following specification:

```
typedef ub4 (*OCISubscriptionNotify) ( void            *pCtx,
                                       OCISubscription *pSubscrHp,
                                       void            *pPayload,
                                       ub4             iPayloadLen,
                                       void            *pDescriptor,
                                       ub4             iMode);
```

The parameters are described as follows:

pCtx (IN)
A user-defined context specified when the callback was registered.

pSubscrHp (IN)
The subscription handle specified when the callback was registered.

pPayload (IN)
The payload for this notification. For this release, only ub1 * (a sequence of bytes) for the payload is supported.

iPayloadLen (IN)
The length of the payload for this notification.

pDescriptor (IN)
The namespace-specific descriptor. Namespace-specific parameters can be extracted from this descriptor. The structure of this descriptor is opaque to the user and its type is dependent on the namespace.

The attributes of the descriptor are namespace-specific. For Advanced Queuing (AQ), the descriptor is `OCI_DTYPE_AQNFY`. For the AQ namespace, the count of notifications received in the group is provided in the notification descriptor. The attributes of `pDescriptor` are:

- Notification Flag (regular = 0, timeout = 1, or grouping notification = 2) - `OCI_ATTR_NFY_FLAGS`
- Queue Name - `OCI_ATTR_QUEUE_NAME`

- Consumer Name - OCI_ATTR_CONSUMER_NAME
- Message Id - OCI_ATTR_NFY_MSGID
- Message Properties - OCI_ATTR_MSG_PROP
- Count of Notifications Received In the Group - OCI_ATTR_AQ_NTFN_GROUPING_COUNT
- The Group, an OCI Collection - OCI_ATTR_AQ_NTFN_GROUPING_MSGID_ARRAY

See Also:

- "OCI and Streams Advanced Queuing" on page 9-41
- "Notification Descriptor Attributes" on page A-56

iMode (IN)

Call-specific mode. Valid value:

- OCI_DEFAULT - executes the default call

Example of OCI Notification Callback for AQ Grouping

Here is an example of how to use the new AQ grouping notification attributes in a notification callback.

```
ub4 notifyCB1(void *ctx, OCISubscription *subscrhp, void *pay, ub4 payl,
              void *desc, ub4 mode)
{
 oratext            *subname;
 ub4                 size;
 OCIColl            *msgid_array = (OCIColl *)0;
 ub4                 msgid_cnt = 0;
 OCIRaw             *msgid;
 void               **msgid_ptr;
 sb4                 num_msgid = 0;
 void               *elemind = (void *)0;
 boolean             exist;
 ub2                 flags;
 oratext            *hexit = (oratext *)"0123456789ABCDEF";
 ub4                 i, j;

 /* get subscription name */
 OCIAttrGet(subscrhp, OCI_HTYPE_SUBSCRIPTION, (void *)&subname, &size,
            OCI_ATTR_SUBSCR_NAME,ctxptr->errhp);

 /* print subscripton name */
 printf("Got notification for %.*s\n", size, subname);
 fflush((FILE *)stdout);

 /* get the #ntfns received in this group */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY, (void *)&msgid_cnt, &size,
            OCI_ATTR_AQ_NTFN_GROUPING_COUNT, ctxptr->errhp);

 /* get the group - collection of msgids */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY, (void *)&msgid_array, &size,
            OCI_ATTR_AQ_NTFN_GROUPING_MSGID_ARRAY, ctxptr->errhp);

 /* get notification flag - regular, timeout, or grouping notification? */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY, (void *)&flags, &size,
            OCI_ATTR_NFY_FLAGS, ctxptr->errhp);
```

```
/* print notification flag */
printf("Flag: %d\n", (int)flags);

/* get group (collection) size */
if (msgid_array)
  checkerr(ctxptr->errhp,
       OCICollSize(ctxptr->envhp, ctxptr->errhp,
       CONST OCIColl *) msgid_array, &num_msgid),
       "Inside notifyCB1-OCICollSize");
else
  num_msgid =0;

/* print group size */
printf("Collection size: %d\n", num_msgid);

/* print all msgids in the group */
for(i = 0; i < num_msgid; i++)
{
  ub4  rawSize;                                          /* raw size    */
  ub1 *rawPtr;                                           /* raw pointer */
    /* get msgid from group */
  checkerr(ctxptr->errhp,
       OCICollGetElem(ctxptr->envhp, ctxptr->errhp,
              (OCIColl *) msgid_array, i, &exist,
              (void **)(&msgid_ptr), &elemind),
       "Inside notifyCB1-OCICollGetElem");
  msgid = *msgid_ptr;
  rawSize = OCIRawSize(ctxptr->envhp, msgid);
  rawPtr = OCIRawPtr(ctxptr->envhp, msgid);

  /* print msgid size */
  printf("Msgid size: %d\n", rawSize);

  /* print msgid in hex format */
  for (j = 0; j < rawSize; j++)
  {                                          /* for each byte in the raw */
    printf("%c", hexit[(rawPtr[j] & 0xf0) >> 4]);
    printf("%c", hexit[(rawPtr[j] & 0x0f)]);
  }
  printf("\n");
}

/* print #ntfns received in group */
printf("Notification Count: %d\n", msgid_cnt);
printf("\n");
printf("************************************************************\n");
fflush((FILE *)stdout);
return 1;
}
```

Notification Procedure

The PL/SQL procedure that will be invoked when there is some activity on the subscription for which interest has been registered, has to be created in the database.

This procedure is typically set through the `OCI_ATTR_SUBSCR_RECPT` attribute of the subscription handle.

See Also:

- See "Subscription Handle Attributes" on page A-48.
- "Oracle Streams AQ PL/SQL Callback" in *Oracle Database PL/SQL Packages and Types Reference* for the PL/SQL procedure specification.

Publish-Subscribe Direct Registration Example

This example shows how system events, client notification, and Advanced Queuing work together to implement publish subscription notification.

The following PL/SQL code creates all objects necessary to support a publish-subscribe mechanism under the user schema `pubsub`. In this code, the Agent `snoop` subscribes to messages that are published at logon events. Note that the user `pubsub` needs `AQ_ADMINISTRATOR_ROLE` and `AQ_USER_ROLE` privileges to use Advance Queuing functionality. The initialization parameter `_SYSTEM_TRIG_ENABLED` must be set to `TRUE` (the default) to enable triggers for system events. Connect as `pubsub` before running the following code:

```
-------------------------------------------------------------
----create queue table for persistent multiple consumers
-------------------------------------------------------------
---- Create or replace a queue table
begin
  DBMS_AQADM.CREATE_QUEUE_TABLE(
  QUEUE_TABLE=>'pubsub.raw_msg_table',
  MULTIPLE_CONSUMERS => TRUE,
  QUEUE_PAYLOAD_TYPE =>'RAW',
  COMPATIBLE => '8.1.5');
end;
/
-------------------------------------------------------------
---- Create a persistent queue for publishing messages
-------------------------------------------------------------
---- Create a queue for logon events
begin
  DBMS_AQADM.CREATE_QUEUE(QUEUE_NAME=>'pubsub.logon',
  QUEUE_TABLE=>'pubsub.raw_msg_table',
  COMMENT=>'Q for error triggers');
end;
/
-------------------------------------------------------------
---- Start the queue
-------------------------------------------------------------
begin
  DBMS_AQADM.START_QUEUE('pubsub.logon');
end;
/
-------------------------------------------------------------
---- define new_enqueue for convenience
-------------------------------------------------------------
create or replace procedure new_enqueue(queue_name  in varchar2,
                                        payload  in raw ,
correlation in varchar2 := NULL,
exception_queue in varchar2 := NULL)
as
  enq_ct     dbms_aq.enqueue_options_t;
  msg_prop   dbms_aq.message_properties_t;
  enq_msgid  raw(16);
```

```
  userdata   raw(1000);
begin
  msg_prop.exception_queue := exception_queue;
  msg_prop.correlation := correlation;
  userdata := payload;
  DBMS_AQ.ENQUEUE(queue_name,enq_ct, msg_prop,userdata,enq_msgid);
end;
/
-------------------------------------------------------------
---- add subscriber with rule based on current user name,
---- using correlation_id
-------------------------------------------------------------
declare
subscriber sys.aq$_agent;
begin
  subscriber := sys.aq$_agent('SNOOP', null, null);
  dbms_aqadm.add_subscriber(queue_name => 'pubsub.logon',
                            subscriber => subscriber,
                            rule => 'CORRID = ''ix'' ');
end;
/
-------------------------------------------------------------
---- create a trigger on logon on database
-------------------------------------------------------------
---- create trigger on after logon
create or replace trigger systrig2
   AFTER LOGON
   ON DATABASE
   begin
     new_enqueue('pubsub.logon', hextoraw('9999'), dbms_standard.login_user);
   end;
/

-------------------------------------------------------------
---- create a PL/SQL callback for notification of logon
---- of user 'ix' on database
-------------------------------------------------------------
----
create or replace procedure plsqlnotifySnoop(
  context raw, reginfo sys.aq$_reg_info, descr sys.aq$_descriptor,
  payload raw, payloadl number)
as
begin
 dbms_output.put_line('Notification : User ix Logged on\n');
end;
/
```

After the subscriptions are created, the client needs to register for notification using callback functions. The following sample code performs the necessary steps for registration. The initial steps of allocating and initializing session handles are omitted here for sake of clarity.

```
...
static ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;

static OCISubscription *subscrhpSnoop = (OCISubscription *)0;
static OCISubscription *subscrhpSnoopMail = (OCISubscription *)0;
static OCISubscription *subscrhpSnoopServer = (OCISubscription *)0;

/* callback function for notification of logon of user 'ix' on database */
```

```
static ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
    void *ctx;
    OCISubscription *subscrhp;
    void *pay;
    ub4 payl;
    void *desc;
    ub4 mode;
{
    printf("Notification : User ix Logged on\n");
  (void)OCIHandleFree((void *)subscrhpSnoop,
            (ub4) OCI_HTYPE_SUBSCRIPTION);
    return 1;
}

static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  ub4 buflen;
  sb4 errcode;

  if (status == OCI_SUCCESS) return;

  switch (status)
  {
  case OCI_SUCCESS_WITH_INFO:
    printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    printf("Error - OCI_NO_DATA\n");
    break;
  case OCI_ERROR:
    OCIErrorGet ((void *) errhp, (ub4) 1, (text *) NULL, &errcode,
            errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
    break;
  case OCI_INVALID_HANDLE:
    printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    printf("Error - OCI_STILL_EXECUTING\n");
    break;
  case OCI_CONTINUE:
    printf("Error - OCI_CONTINUE\n");
    break;
  default:
    printf("Error - %d\n", status);
    break;
  }
}

static void initSubscriptionHn (subscrhp,
                        subscriptionName,
                        func,
                        recpproto,
```

```
                        recpaddr,
                        recppres)
OCISubscription **subscrhp;
  char * subscriptionName;
  void * func;
  ub4 recpproto;
  char * recpaddr;
  ub4 recppres;
{
    /* allocate subscription handle */
    (void) OCIHandleAlloc((void *) envhp, (void **)subscrhp,
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (void **) 0);

    /* set subscription name in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) subscriptionName,
        (ub4) strlen((char *)subscriptionName),
        (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

    /* set callback function in handle */
    if (func)
      (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
          (void *) func, (ub4) 0,
          (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

    /* set context in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) 0, (ub4) 0,
       (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

    /* set namespace in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) &namespace, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

    /* set receive with protocol in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) &recpproto, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_RECPTPROTO, errhp);

    /* set recipient address in handle */
    if (recpaddr)
      (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
          (void *) recpaddr, (ub4) strlen(recpaddr),
          (ub4) OCI_ATTR_SUBSCR_RECPT, errhp);

    /* set receive with presentation in handle */
    (void) OCIAttrSet((void *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (void *) &recppres, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_RECPTPRES, errhp);

    printf("Begining Registration for subscription %s\n", subscriptionName);
    checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,
        OCI_DEFAULT));
   printf("done\n");

}
```

```
int main( argc, argv)
int    argc;
char * argv[];
{
    OCISession *authp = (OCISession *) 0;

/*************************************************
Initialize OCI Process/Environment
Initialize Server Contexts
Connect to Server
Set Service Context
*************************************************/

/* Registration Code Begins */
/* Each call to initSubscriptionHn allocates
   and Initialises a Registration Handle */

/* Register for OCI notification */
    initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle*/
    (char*) "PUBSUB.LOGON:SNOOP", /* subscription name */
                                  /*<queue_name>:<agent_name> */
        (void*)notifySnoop,  /* callback function */
        OCI_SUBSCR_PROTO_OCI, /* receive with protocol */
        (char *)0, /* recipient address */
        OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

/* Register for email notification */
    initSubscriptionHn(    &subscrhpSnoopMail,  /* subscription handle */
     (char*) "PUBSUB.LOGON:SNOOP",              /* subscription name */
                                                /* <queue_name>:<agent_name> */
        (void*)0, /* callback function */
        OCI_SUBSCR_PROTO_MAIL, /* receive with protocol */
        (char*)  "longying.zhao@oracle.com", /* recipient address */
        OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

/* Register for server to server notification */
    initSubscriptionHn(    &subscrhpSnoopServer, /* subscription handle */
       (char*)  "PUBSUB.LOGON:SNOOP",            /* subscription name */
                                                 /* <queue_name>:<agent_name> */
        (void*)0, /* callback function */
        OCI_SUBSCR_PROTO_SERVER, /* receive with protocol */
         (char*) "pubsub.plsqlnotifySnoop", /* recipient address */
        OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

    checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) OCI_DEFAULT));

/*************************************************
The Client Process does not need a live Session for Callbacks
End Session and Detach from Server
*************************************************/

    OCISessionEnd ( svchp,  errhp, authp, (ub4) OCI_DEFAULT);

    /* detach from server */
    OCIServerDetach( srvhp, errhp, OCI_DEFAULT);

   while (1)    /* wait for callback */
    sleep(1);
}
```

If user IX logs on to the database, the client is notified by e-mail, and the callback function notifySnoop is called. An e-mail notification will be sent to the address xyz@company.com and the PL/SQL procedure plsqlnotifySnoop will also be called in the database.

Publish-Subscribe LDAP Registration Example

The following code fragment illustrates how to do LDAP registration. Please read all the program comments:

```
...

  /* TO use LDAP registration feature, OCI_EVENTS | OCI_EVENTS |OCI_USE_LDAP must
  /*   be set in OCIInitialize: */
  (void) OCIInitialize((ub4) OCI_EVENTS|OCI_OBJECT|OCI_USE_LDAP, (void *)0,
                       (void * (*)(void *, size_t)) 0,
                       (void * (*)(void *, void *, size_t))0,
                       (void (*)(void *, void *)) 0 );

...

  /* set LDAP attributes in the environment handle */

  /* LDAP host name */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"yow", 3,
                    OCI_ATTR_LDAP_HOST, (OCIError *)errhp);

  /* LDAP server port */
  ldap_port = 389;
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)&ldap_port,
                    (ub4)0, OCI_ATTR_LDAP_PORT, (OCIError *)errhp);

  /* bind DN of the client, normally the enterprise user name */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"cn=orcladmin",
                    12, OCI_ATTR_BIND_DN, (OCIError *)errhp);

  /* password of the client */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"welcome",
                    7, OCI_ATTR_LDAP_CRED, (OCIError *)errhp);

  /* authentication method is "simple", username/password authentication */
  ldap_auth = 0x01;
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)&ldap_auth,
                    (ub4)0, OCI_ATTR_LDAP_AUTH, (OCIError *)errhp);

  /* adminstrative context: this is the DN above cn=oraclecontext */
  (void) OCIAttrSet((void *)envhp, OCI_HTYPE_ENV, (void *)"cn=acme,cn=com",
                    14, OCI_ATTR_LDAP_CTX, (OCIError *)errhp);

...

  /* retrieve the LDAP attributes from the environment handle */

  /* LDAP host */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                    &szp,  OCI_ATTR_LDAP_HOST,  (OCIError *)errhp);

  /* LDAP server port */
  (void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&intval,
```

```
                0,  OCI_ATTR_LDAP_PORT,  (OCIError *)errhp);

/* client binding DN */
(void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                &szp,  OCI_ATTR_BIND_DN,  (OCIError *)errhp);

/* client password */
(void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                &szp,  OCI_ATTR_LDAP_CRED,  (OCIError *)errhp);

/* adminstrative context */
(void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&buf,
                &szp,  OCI_ATTR_LDAP_CTX,  (OCIError *)errhp);

/* client authentication method */
(void) OCIAttrGet((void *)envhp, OCI_HTYPE_ENV, (void *)&intval,
                0,  OCI_ATTR_LDAP_AUTH,  (OCIError *)errhp);

...

/* to set up the server DN descriptor in the subscription handle */

/* allocate a server DN descriptor, dn is of type "OCIServerDNs **",
   subhp is of type "OCISubscription **" */
(void) OCIDescriptorAlloc((void *)envhp, (void **)dn,
                     (ub4) OCI_DTYPE_SRVDN, (size_t)0, (void **)0);

/* now *dn is the server DN descriptor, add the DN of the first database
   that we want to register */
(void) OCIAttrSet((void *)*dn, (ub4) OCI_DTYPE_SRVDN,
                (void *)"cn=server1,cn=oraclecontext,cn=acme,cn=com",
                42, (ub4)OCI_ATTR_SERVER_DN, errhp);
/* add the DN of another database in the descriptor */
(void) OCIAttrSet((void *)*dn, (ub4) OCI_DTYPE_SRVDN,
                (void *)"cn=server2,cn=oraclecontext,cn=acme,cn=com",
                42, (ub4)OCI_ATTR_SERVER_DN, errhp);

/* set the server DN descriptor into the subscription handle */
(void) OCIAttrSet((void *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
               (void *) *dn, (ub4)0, (ub4) OCI_ATTR_SERVER_DNS, errhp);

...

/* now we will try to get the server DN information from the subscription
   handle */

/* first, get the server DN descriptor out */
(void) OCIAttrGet((void *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                (void *)dn, &szp, OCI_ATTR_SERVER_DNS, errhp);

/* then, get the number of server DNs in the descriptor */
(void) OCIAttrGet((void *) *dn, (ub4)OCI_DTYPE_SRVDN, (void *)&intval,
                &szp, (ub4)OCI_ATTR_DN_COUNT, errhp);

/* allocate an array of char * to hold server DN pointers returned by
   oracle */
  if (intval)
  {
    arr = (char **)malloc(intval*sizeof(char *));
    (void) OCIAttrGet((void *)*dn, (ub4)OCI_DTYPE_SRVDN, (void *)arr,
```

```
                        &intval, (ub4)OCI_ATTR_SERVER_DN, errhp);
    }

  /* OCISubscriptionRegister() calls have two modes: OCI_DEFAULT and
     OCI_REG_LDAPONLY. If OCI_DEFAULT is used, there should be only one
     server DN in the server DN descriptor. The registration request will
     be sent to the database. If a database connection is not available,
     the registration request will be detoured to the LDAP server. On the
     other hand, if mode OCI_REG_LDAPONLY is used the registration request
     will be directly sent to LDAP. This mode should be used when there are
     more than one server DNs in the server DN descriptor, or we are sure
     that a database connection is not available.

     In this example, two DNs are entered; so we should use mode
     OCI_REG_LDAPONLY in register. */
  OCISubscriptionRegister(svchp, subhp, 1, errhp, OCI_REG_LDAPONLY);

  ...

  /* as OCISubscriptionRegister(), OCISubscriptionUnregister() also has
     mode OCI_DEFAULT and OCI_REG_LDAPONLY. The usage is the same. */

  OCISubscriptionUnRegister(svchp, *subhp, errhp, OCI_REG_LDAPONLY);
}
...
```