Skip Headers

Oracle Application Server TopLink Application Developer's Guide 10*g* (9.0.4)
Part Number B10313-01

Home Solution Area Contents Index

4 Sessions

Sessions are a key component of the Oracle Application Server TopLink application--they provide OracleAS TopLink with access to the database. Sessions enable you to execute queries, and they return persistent objects and other results for client applications. This chapter introduces OracleAS TopLink sessions, and describes:

Introduction to Session Concepts

A session represents the connection between an application and the relational database that stores its persistent objects. OracleAS TopLink provides different session classes, each optimized for different design requirements and data access strategies. OracleAS TopLink session types range from a simple database session that gives one user one connection to the database, to the session broker that provides access to several databases for multiple clients.

To understand the OracleAS TopLink session, you must be familiar with several session concepts.

sessions.xml File

In most cases, the developer pre configures sessions for the application in a session configuration file. This file, known as the `sessions.xml` file, is an Extensible Markup Language

(XML) file that contains all sessions that are associated with the application. The `sessions.xml` file can contain any number of sessions and session types.

Session Types

Several session types each provide a particular set of functionality to the application.

Server Session

A server session is the most common OracleAS TopLink session type, because it supports the three-tier architectures that are common to enterprise applications. Server sessions manage the server side of client-server communications. They work together with the client session to provide complete client-server communication.

The server session provides shared resources to a multithreaded environment, including a shared cache and connection pools. The server session also provides transaction isolation.

For more information about the server session, see "Server Session and Client Session".

Client Session

A client session is a client-side communications mechanism that works together with the server session to provide the client-server connection. Each client session serves one client.

For more information about the client session, see "Server Session and Client Session".

Remote Session

A remote session offers database access to clients that do not reside on the OracleAS TopLink Java virtual machine (JVM). The remote session connects to a client session, which, in turn, connects to the server session.

For more information, see "Remote Session".

Database Session

A database session is a unique session type because it provides both client and server communications. It is a relatively simple session type that supports only a single client and a single database connection. The database session is not scalable; however, if you have an application with a single client that requires only one database connection, the database session is usually your best choice.

For more information, see "Database Session".

Session Broker

The OracleAS TopLink session broker is a mechanism that enables client applications to communicate with multiple databases. A session broker makes multiple database access transparent to the client.

For more information, see "Session Broker".

Session Manager

When a client application requires a session, it requests the session from the OracleAS TopLink session manager. The two main functions of the session manager are to instantiate OracleAS TopLink sessions for the server, and to hold the sessions for the life of the application. The session manager instantiates database sessions, server sessions, or session brokers based on the configuration information in the `sessions.xml` file.

The session manager instantiates sessions as follows:

1. The client application request a session by name.
2. The session manager looks up the session name in the `sessions.xml` file. If the session name exists, the session manager instantiates the specified session; otherwise, it raises an exception.
3. After instantiation, the session remains viable until you shut down the application.

Connection Pool

A connection pool is a collection of reusable database connections. OracleAS TopLink manages these connections for the application, provides connections to processes as needed, and returns connections to the pool when the process is complete. When it is returned to the pool, the connection is available for other processes.

A properly configured connection pool significantly improves performance.

For more information about configuring connection pools, see "Working with Connection Pools" in the *Oracle Application Server TopLink Mapping Workbench User's Guide*.

Caching

OracleAS TopLink sessions provide an object cache. This cache, known as the *session cache*, retains information about objects that are read from or written to the database, and is a key element for improving the performance of an OracleAS TopLink application.

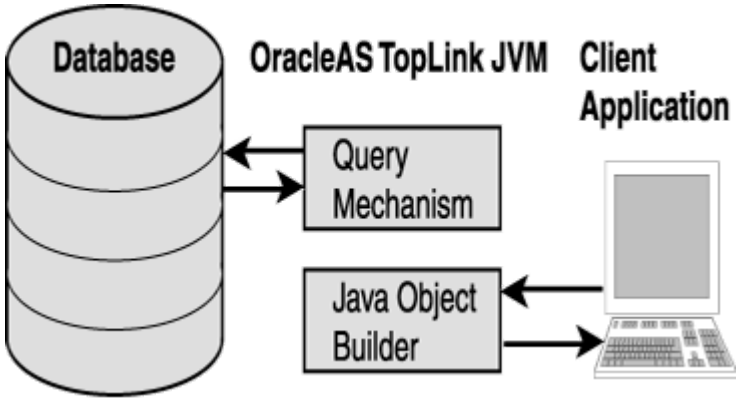
Profiling

OracleAS TopLink profiling enables you to identify performance bottlenecks in your application. When enabled, the profiler logs a summary of the performance statistics for every query that the application executes.

Session Architectures

A session in an OracleAS TopLink application includes a query mechanism that interacts with the database, and an object construction mechanism that builds objects from the data that is stored in the database. The data interaction and object construction components both reside on a JVM. A client application uses these mechanisms to query the database and retrieve objects.

Figure 4-1 Simple OracleAS TopLink Session Architecture

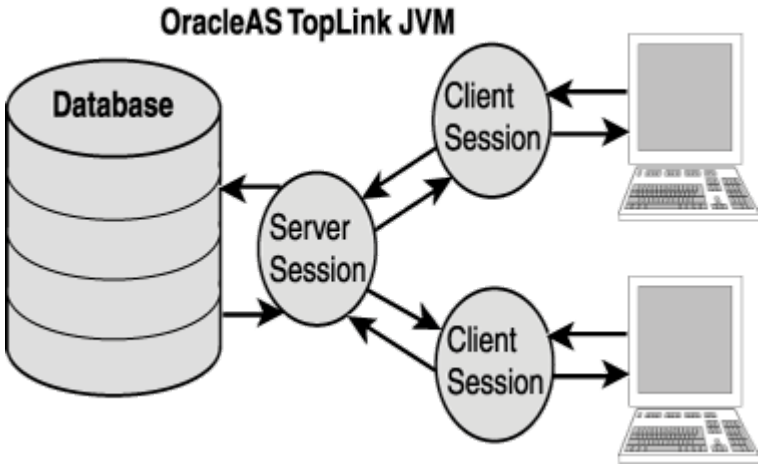


Text description of the illustration sessarch.gif

Server Session

A server session provides a connection with the database, and makes the extracted data available to one or more client session (either client session or remote sessions). A server session usually appears as part of an OracleAS TopLink three-tier architecture. It uses a JDBC connection pool configured to provide a query mechanism to clients. Client applications communicate with the server session through a client session.

Figure 4-2 Typical OracleAS TopLink Server Session with Client Session Architecture



Text description of the illustration sessclie.gif

For more information about the server session, see "Server Session and Client Session".

Client Session

A client session communicates with the server session on behalf of the client application (see Figure 4-2). A server session creates client sessions on request, and the client sessions share an object cache.

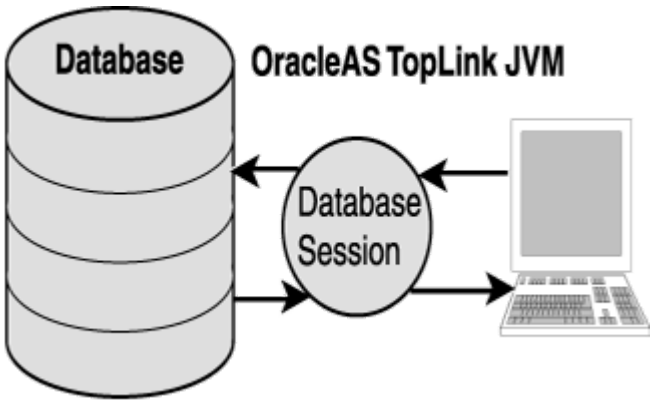
Together, the client session and server session provide a three-tier architecture that you can scale easily, by adding more client sessions. Because of this scaleability, we recommend you use the three-tier architecture to build your OracleAS TopLink applications.

For more information about the client session, see "Server Session and Client Session".

Database Session

A database session provides a client application with a single JDBC database connection, for simple, standalone applications in which a single connection services all database requests for one user.

Figure 4-3 OracleAS TopLink Database Session Architecture



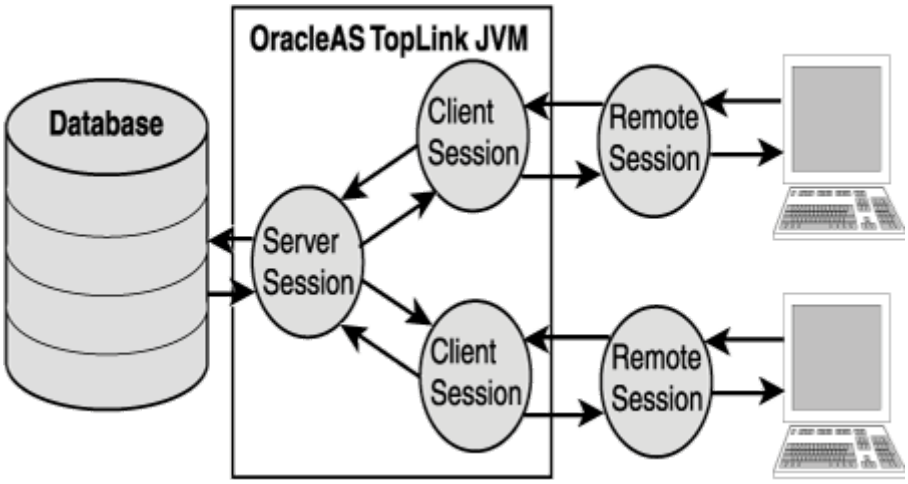
Text description of the illustration dbsess.gif

For more information about the database session, see "Database Session".

Remote Session

A remote session is a client-side session that resides on the client rather than the OracleAS TopLink JVM. The remote session does not replace the client session; rather, a remote session requires a client session to communicate with the server session. A remote session can also communicate directly with a database session.

Figure 4-4 Typical OracleAS TopLink Server Session with Remote Session Architecture



Text description of the illustration remtsess.gif

The remote session provides a full OracleAS TopLink session, complete with a session cache, on the client system. OracleAS TopLink manages the remote session cache and enables client applications to execute operations on the OracleAS TopLink JVM.

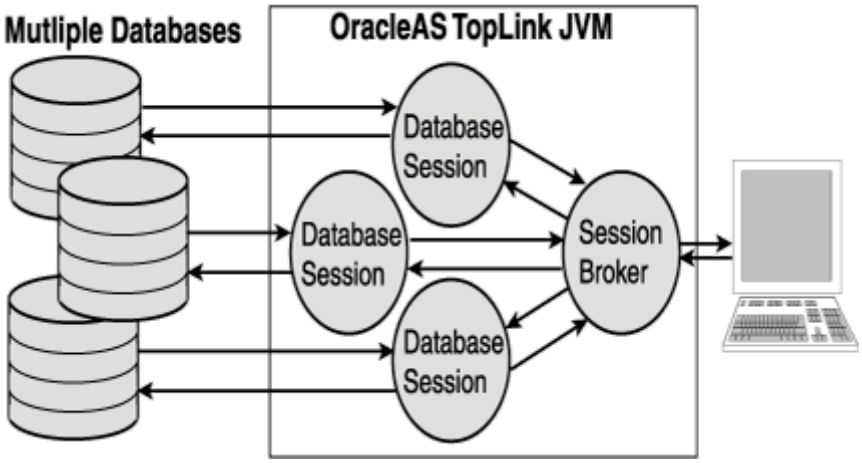
For more information about the remote session, see "Remote Session".

Session Broker

The OracleAS TopLink session broker enables client applications to view several databases through a single session. If you store objects in your application on multiple databases, the session broker, which provides seamless communication for client applications, enables the client to view multiple databases as if they are a single database.

The session broker connects to the databases through either a database session or a server session.

Figure 4-5 OracleAS TopLink Session Broker with Server Session Architecture



Text description of the illustration sessbrok.gif

For more information about the session broker, see "Session Broker".

Configuring Sessions with the sessions.xml File

OracleAS TopLink provides two ways to preconfigure your sessions: you can export and compile Java source code from the OracleAS TopLink Mapping Workbench, or use the OracleAS TopLink Sessions Editor to build a session configuration file, the `sessions.xml` file. For the following reasons, we recommend you use the `sessions.xml` file to deploy an OracleAS TopLink application:

- It is easy to create and maintain in the OracleAS TopLink Sessions Editor.
- It is easy to troubleshoot.
- It provides access to most session configuration options.
- It offers excellent flexibility, including the ability to modify deployed applications.

This section describes the `sessions.xml` file and illustrates the options that are available when you build the file. This section discusses editing the file manually, but the simplest way to build the `sessions.xml` file is to use the OracleAS TopLink Sessions Editor in the OracleAS TopLink Mapping Workbench.

This section explains how to configure the `sessions.xml` file, and includes discussions on:

- Navigating the sessions.xml File
- XML Header
- toplink-configuration Element

- session Element
- session-broker Element
- JTA Configuration

For more information about creating configuration files in the OracleAS TopLink Mapping Workbench, see "Understanding the OracleAS TopLink Sessions Editor" in the *Oracle Application Server TopLink Mapping Workbench User's Guide*.

Navigating the sessions.xml File

The `sessions.xml` file's Document Type Definition (DTD) defines the file structure. If you use the OracleAS TopLink Sessions Editor, you need not concern yourself with that structure. However, if you do create or edit the file, you must understand its structure.

The main structure of the `sessions.xml` file is the `toplink-configuration` element. This element includes all session configuration options. Within the `toplink-configuration` element, you configure sessions and session brokers. The session broker contains only sessions defined in the `sessions.xml` file; the bulk of session configuration occurs within the `session` element.

Example 4-1 offers a navigational view of the `sessions.xml` file, illustrating the file's structure:

Example 4-1 Navigating the sessions.xml File

```
<toplink-configuration>
   <session>
     <name>
     <project-class> or <project-xml>
     <session-type>
     <login>
        [Login Options including Sequencing and Cache Sysnchronization]
        <uses-external-connection-pool>
        <uses-external-transaction-controller>
     </login>
     <event-listener-class>
     <profiler-class>
     <data-source>
     <external-transaction-controller-class>
     <exception-handler-class>
     <connection-pool>
        [Connection Pool Options]
     </connection-pool>
     <enable-logging>
        [Logging Options]
     </enable-logging>>
   </session>
</toplink-configuration>
```

XML Header

The `sessions.xml` file begins with a header section that describes the file, and specifies the location of the DTD for file validation.

If you use third-party parsers with the `sessions.xml` file, be aware that some parsers require a fully qualified path to the DTD in the XML header. If you are using one of these parsers, include the full path to the DTD in the system identifier, as follows:

```
<!DOCTYPE toplink-configuration PUBLIC "-//Oracle Corp.//DTD TopLink Sessions
9.0.4//EN" "file://<ORACLE_HOME>/toplink/config/dtds/sessions_9_0_4.dtd">
```

toplink-configuration Element

The `toplink-configuration` element is the root XML element for the `sessions.xml` file. It encapsulates the rest of the session configuration information.

Example 4-2 The toplink-configuration Element

```
<toplink-configuration>
    ...
    //Session configuration information
    ...
</toplink-configuration>
```

session Element

The `session` element contains configuration information for an OracleAS TopLink session. It includes several tags that specify the options for the session. The `sessions.xml` file normally contains at least one `session` element, and can include several elements if the application requires it.

The `session` element supports the configuration tags listed in Table 4-1.

Table 4-1 Tags Within the Session Element

Tag	Description
`name`	Specifies the name of the session. Assign a unique name to each session in the `sessions.xml` file to enable the session manager to retrieve it correctly. The `name` tag is mandatory.
`project-class`	Specifies the name of the class that contains the OracleAS TopLink project metadata. Use this tag (and not the `project-xml` tag) to deploy a project that uses exported and compiled Java code. Specify the fully qualified Java class name, but do not include the `.class` or `.java` extension.
`project-xml`	Specifies the name of the XML file that contains the OracleAS TopLink project metadata. Use this tag (and not the `project-class` tag) to deploy your project that uses an exported XML file. Specify the fully qualified file name, including the .xml extension.

Example 4-3 Using a Project Class Element

```
<toplink-configuration>
    <session>
        <name>mysession</name>
        <project-class>com.mycompany.MyProject</project-class>
        ...
    </session>
</toplink-configuration>
```

Example 4-4 Using the project.xml File

```
<toplink-configuration>
    <session>
        <name>mysession</name>
        <project-xml>C:/myproject/myproject.xml</project-xml>
        ...
    </session>
</toplink-configuration>
```

In addition to the preceding tags, the `session` element includes several tags that contain session configuration information:

- session-type Element
- login Element
- event-listener-class Element
- cache-synchronization-manager Element
- profiler-class Element
- external-transaction-controller-class Element
- exception-handler-class Element
- connection-pool Element
- enable-logging Element

session-type Element

The `session-type` element appears inside of a `session` element and specifies the session type with the tags listed in Table 4-2.

Table 4-2 Tags Within the Session-Type Element

Tag	Description
`session-type`	Specifies the type of OracleAS TopLink session the `SessionManager` will instantiate. Valid options include `server-session` and `database-session`. The `session-type` tag is mandatory.
`server-session`	In the `session-type` element, indicates that the `SessionManager` instantiates and returns the named session as a `ServerSession` (Server).
`database-session`	In the `session-type` element, indicates that the `SessionManager` instantiates and returns the named session as a `DatabaseSession`.

Example 4-5 Defining a Server Session

```
<session>
    <name>myServerSession</name>
    <project-class>com.mycompany.MyProject</project-class>
    <session-type>
```

```
        <server-session/>
    </session-type>
    ...
</session>
```

Example 4-6 Defining a Database Session

```
<session>
    <name>myDatabaseSession</name>
    <project-class>com.mycompany.MyProject</project-class>
    <session-type>
        <database-session/>
    </session-type>
    ...
</session>
```

login Element

The `login` element tags listed in Table 4-3 are optional for the session. If you do not include the `login` element in the `sessions.xml` file, set a default login in the OracleAS TopLink Mapping Workbench.

Table 4-3 Basic Configuration Tags Within the Login Element

Tag	Description
`license-path`	Specifies the license path for pre-TopLink 4.6 licensing. Because OracleAS TopLink no longer requires this tag, OracleAS TopLink does not process this element. If you are using the `sessions.xml` file from an OracleAS TopLink version that required a licence file, this tag will not prevent the `sessions.xml` file from running under the current version of OracleAS TopLink, but you should consider rebuilding your `sessions.xml` file. Note: If you are using a `sessions.xml` file from an older version of OracleAS TopLink, you can delete this tag.
`driver-class`	Specifies the JDBC driver class to use to log in to the database. The `driver-class` tag is optional and is not required when you implement the `data-source` tag.
`connection-url`	Specifies the JDBC connection URL for the database. This tag is optional. Do not use the `connection-url` tag if you implement the `data-source` tag.
`data-source`	Specifies the datasource name if you are using a JNDI datasource. This tag is optional. Do not use the `data-source` tag if you implement the `connection-url` and `driver-class` tag.

Tag	Description
`platform-class`	Specifies the OracleAS TopLink platform class for the session. This tag is optional. For more information about platform classes, see "SDK Platform and Sequencing".
`user-name`	The user name to log in to the database. The `user-name` tag is optional and is not required if you use a datasource.
`password`	The password to log in to the database. The `password` tag is optional and is not required if you use a datasource.
`encrypted-password`	The password of the user name used to log into the database. The `<encrypted-password>` tag.
`encryption-class- name`	When you use an encrypted password, select the specific encryption class. The `<encryption-class-name>` tag.

Example 4-7 Basic Configuration Using JDBC

```
<session>
  <name>myServerSession</name>
  <project-class>com.mycompany.MyProject</project-class>
  <session-type>
    <server-session/>
  </session-type>
  <login>
    <license-path>C:/myproject/license/</license-path>
    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <connection-url>jdbc:oracle:thin@dbserver:1521:dbname</connection-url>
    <platform-class>oracle.toplink.internal.databaseaccess.OraclePlatform</platform-class>
    <user-name>scott</user-name>
    <password>tiger</password>
  </login>
...
</session>
```

Example 4-8 Basic Configuration Using a Datasource

```
<session>
  <name>myServerSession</name>
  <project-class>com.mycompany.MyProject</project-class>
  <session-type>
    <server-session/>
  </session-type>
  <login>
    <data-source>jdbc/MyApplicationDS</data-source>
    <platform-class>oracle.toplink.internal.databaseaccess.OraclePlatform</platform-class>
  </login>
...
</session>
```

Optional Login Tags

The `login` element offers several optional tags that enable you to customize your session login.

Optional tags the `login` element offers include:

- `encryption-class-name`: Specifies the name of the custom class used to encrypt and decrypt the password. The `encryption-class-name` must be fully qualified and the class must be on the class path.
- `encrypted-password`: Specifies the encrypted password.

Other optional `login` tags accept TRUE or FALSE as valid values. Table 4-4 describes these tags.

Table 4-4 Optional Tags Within the Login Element

Tag	Description
`should-bind-all-parameters`	Enables parameter binding for all parameters. Use parameter binding with statement caching. The default value is FALSE. For more information about Parameter Binding, see "Binding and Parameterized SQL".
`should-cache-all-statements`	Enables statement caching. The default value is FALSE. Statement caching requires you to set the `should-bind-all-parameters` tag to TRUE.
`uses-byte-array-binding`	Specifies whether OracleAS TopLink uses binding for byte arrays. The default value is FALSE.
`uses-string-binding`	Specifies whether OracleAS TopLink uses binding for String objects. The default value is FALSE.
`uses-streams-for-binding`	Specifies whether OracleAS TopLink uses streams for binding byte array parameters. The default value is FALSE.
`should-force-field-names-to-uppercase`	Specifies whether OracleAS TopLink converts field names to uppercase when generating SQL. The default value is FALSE.

Tag	Description
`should-optimize-data-conversion`	Specifies whether the session should optimize driver-level data conversion. The default value is TRUE.
`should-trim-strings`	Specifies whether OracleAS TopLink removes any trailing white spaces from the end of strings. The default value is TRUE.
`uses-batch-writing`	Specifies whether the session uses batch writing to write to the database. The default value is FALSE.
`uses-jdbc20-batch-writing`	Specifies whether the session's database connection(s) uses JDBC 2.0 batch writing or OracleAS TopLink batch writing. The default value is TRUE. If you enable this option, enable the `uses-batch-writing` option as well.
`uses-external-connection-pool`	Specifies whether the session uses external connection pooling. The default value is FALSE.
`uses-native-sql`	Specifies whether the session uses database-specific SQL grammar. The default value is FALSE.
`uses-external-transaction-controller`	Specifies whether the session uses an external transaction controller. The default value is FALSE.
`non-jts-connection-url`	Specifies the URL for sequencing connection pooling. Used in conjunction with the `non-jts-datasource` tag when you set the `uses-sequence-connection-pool` tag to TRUE.
`non-jts-datasource`	Specifies the non-JTS datasource for the sequencing connection pool. Used in conjunction with the `non-jts-connection-url` tag when you set the `uses-sequence-connection-pool` tag to TRUE.
`uses-sequence-connection-pool`	Specifies whether the session creates and uses a separate connection pool for sequencing. The default value is FALSE. If you set this element to TRUE, you must also configure the `non-jts-connection-url` and `non-jts-datasource` tags.

Sequencing Elements

You can configure sequencing as part of the session login, although it is not a requirement. If you do not configure sequencing in the `sessions.xml` file, then the application uses the configuration that is specified in the OracleAS TopLink Mapping Workbench project.

Configure sequencing in the `sessions.xml` file when you want to use custom sequencing for a given session.

Table 4-5 lists the elements you use to configure sequencing in the `sessions.xml` file. All these elements are optional.

Table 4-5 Optional Sequencing Configuration Tags Within Login

Tag	Description
`uses-native-sequencing`	Specifies whether the session uses native sequencing. This tag accepts TRUE or FALSE as values. The default is FALSE. Note that not all database platforms support native sequencing.
`sequence-preallocation-size`	Specifies the sequence preallocation size. If you use native sequencing, this value must match the sequence preallocation size set on your database. The default value is 50.
`sequence-table`	For table sequencing, specifies the name of the sequencing table. The default name is `SEQUENCE`.
`sequence-name-field`	For table sequencing, specifies the column in the sequencing table that contains the names of the sequenced objects. The default name is `SEQ_NAME`.
`sequence-counter-field`	For table sequencing, specifies the column in the sequence table that stores the current sequence count for each sequenced object. The default name is `SEQ_COUNT`.

For more information, see "Sequencing".

Example 4-9 Configuring Native Sequencing

```
<session>
    <login>
...
        <uses-native-sequencing>true</uses-native-sequencing>
        <sequence-preallocation-size>50</sequence-preallocation-size>
    </login>
...
</session>
```

Example 4-10 Configuring Table-Based Sequencing

```
<session>
...
    <login>
        <uses-native-sequencing>false</uses-native-sequencing>
        <sequence-table>SEQUENCE</sequence-table>
        <sequence-name-field>SEQ_NAME</sequence-name-field>
        <sequence-counter-field>SEQ_COUNT</sequence-counter-field>
    </login>
...
</session>
```

cache-synchronization-manager Element

You configure cache synchronization as part of the `login`. Use the `cache-synchronization-manager` element and the tags listed in Table 4-6 to configure cache-synchronization for your application.

Table 4-6 Cache Synchronization Manager Configuration Tags

Tag	Description
`clustering-service`	Specifies the class name of the clustering service. This tag is required for cache synchronization.
`multicast-port`	Specifies the port for listening for connection messages over IP multicast. Ensure that all servers in your OracleAS TopLink cache synchronization group use the same multicast port. This tag is required only if you also use the `multicast-group-address` element. The default value is 6018.
`multicast-group-address`	Specifies the IP address for sending connection messages over IP multicast. Ensure that all servers in your OracleAS TopLink cache synchronization group use the same multicast address. This tag is required only if you also use the `multicast-port` element. The default value is 226.18.6.18.
`packet-time-to-live`	Specifies the number of network hops that cache synchronization discovery packets traverse. This optional tag defaults to 2.
`is-asynchronous`	Specifies whether cache synchronization is performed asynchronously (TRUE) or synchronously (FALSE). This optional tag defaults to TRUE.

Tag	Description
`should-remove-connection-on-error`	Specifies whether OracleAS TopLink removes a remote connection if a communications exception occurs with a remote server. This optional tag defaults to FALSE.
`jndi-user-name`	Specifies the user name to use for binding the Cache Synchronization Manager into JNDI. Use this tag to support JNDI in non application server applications. This optional tag requires the `jndi-password` tag.
`jndi-password`	Specifies the password to use for binding the cache synchronization manager into JNDI. Use this tag to support JNDI in non application server applications. This optional tag requires the `jndi-user-name` tag.
`jms-topic-connection-factory-name`	Specifies the topic connection factory name for JMS cache synchronization. This tag is required only when you use JMS cache synchronization.
`jms-topic-name`	Specifies the topic name for JMS cache synchronization. This tag is required only when you use JMS cache synchronization.
`naming-service-initial-context-factory-name`	Specifies the initial context factory for accessing JNDI. Use this tag only if OracleAS TopLink encounters difficulties connecting to JNDI or JMS.
`naming-service-url`	Specifies the URL of the naming service that supports cache synchronization. The value for this element depends on how you implement cache synchronization: • For JNDI clustering services, this is the scheme, host IP address, and port of the JNDI service. • For the RMI clustering service, this is the host IP address and port of the RMI registry. This optional tag may resolve problems that occur when you implement cache synchronization inside an application server with a JNDI clustering service. If you do not encounter any problems, do not use this tag.

Example 4-11 Using the Cache Synchronization Manager

```
〈session〉
...
  〈login〉
```

```
    <cache-synchronization-manager>
      <clustering-service>oracle.toplink.remote.rmi.RMIClusteringService</clustering-servi
ce>
      <multicast-port>6020</multicast-port>
      <multicast-group-address>226.18.6.18</multicast-group-address>
      <is-asynchronous>true</is-asynchronous>
      <should-remove-connection-on-error>true</should-remove-connection-on-error>
      <naming-service-url>localhost:1099</naming-service-url>
    </cache-synchronization-manager>
  </login>
...
</session>
```

event-listener-class Element

If your applications need to know when session events take place, use event listeners to register for event notification. Event listeners can be configured in the `sessions.xml` file.

The `event-listener-class` tag enables you to configure listener classes that either implement the `oracle.toplink.sessions.SessionEventListener` interface, or extend the `oracle.toplink.sessions.SessionEventAdapter` class. Configure multiple event listener classes by including multiple `event-listener-class` tags and specifying the implementing class name for each tag.

OracleAS TopLink automatically registers event listeners in the `sessions.xml` file with the session event manager.

For more information, see "Customizing Session Events".

Example 4-12 Setting the Event Listener Class in Code

```
package examples;
import oracle.toplink.sessions.*;
public class MyEventListener extends SessionEventAdapter {
    public void preLogin(SessionEvent event) {
        Session session = event.getSession();
        /* custom code goes here */
    }
}
```

Example 4-13 Setting the Event Listener Class in the sessions.xml File

```
<session>
    ...
    <event-listener-class>examples.MyEventListener</event-listener-class>
    ...
</session>
```

OracleAS TopLink registers the `examples.MyEventListener` class with the session event manager for the session. OracleAS TopLink invokes the `MyEventListener` class `preLogin` method when the `preLogin` event occurs on the session.

profiler-class Element

OracleAS TopLink provides a profiler to optimize your application and identify performance bottlenecks. To implement the performance profiler, use the `profiler-class` tag to include the performance profiler in your session.

Example 4-14 Implementing the Performance Profiler in the sessions.xml File

```
<session>
    ...
    <profiler-class>oracle.toplink.tools.profiler.PerformanceProfiler</profiler-class>
    ...
</session>
```

The `profiler-class` tag supports any class that implements the `oracle.toplink.sessions.SessionProfiler` interface. Because of this, you can build your own profiler and add it to your session--provided that your profiler implements the `oracle.toplink.sessions.SessionProfiler` interface.

Note:

You can implement only one profiler a session.

---

external-transaction-controller-class Element

If your system includes external transactions (under JTA, for example), specify an OracleAS TopLink external transaction controller using the `external-transaction-controller-class` tag.

To use an external transaction controller, specify the following in the session login:

- The external transaction controller
- A datasource on the session
- An external connection pool

Example 4-15 Configuring the External Transaction Controller

```
<session>
    ...
    <login>
        ...
        <uses-external-transaction-controller>true</uses-external-transaction-controller>
        <data-source>jdbc/MyApplicationDS</data-source>
        <uses-external-connection-pool>true</uses-external-connection-pool>
        ...
    </login>
    <external-transaction-controller-class>oracle.toplink.jts.oracle9i.Oracle9iJTSExternal
TransactionController</external-transaction-controller-class>
    ...
</session>
```

exception-handler-class Element

The `exception-handler-class` tag specifies a class that handles exceptions for the session. This tag accepts any class that implements the `oracle.toplink.exceptions.ExceptionHandler`.

Example 4-16 Configuring the Exception Handler in Code

```
package examples;
import oracle.toplink.exceptions.*;
public class MyExceptionHandler implements ExceptionHandler {
   public Object handleException(RuntimeException exception) {
      /*custom code goes here */
   }
}
```

Example 4-17 Configuring the Exception Handler in the sessions.xml File

```
<session>
  ...
  <exception-handler-class>examples.MyExceptionHandler</exception-handler-class>
  ...
</session>
```

connection-pool Element

You can explicitly configure a single connection pool or multiple connection pools for your OracleAS TopLink application with the `connection-pool` element in the `sessions.xml` file. If you do not configure a connection pool for a session, then the session uses the default connection pool that you defined for the project.

Define a login for each `connection-pool` that you define manually. Table 4-7 lists the elements you use to configure the `connection-pool` element in the `sessions.xml` file.

For more information about configuring the connection pool for the project, see "Working with Connection Pools" in the *Oracle Application Server TopLink Mapping Workbench User's Guide*.

For more information about configuring a login, see "login Element".

Table 4-7 Connection Pool Element Tags

Tag	Description
`is-read-connection-pool`	Specifies whether the connection pool contains read connections (true) - (non-transactional) or for write connections (false) - (transactional). The `is-read-connection-pool` tag is mandatory, and accepts TRUE or FALSE as values.
`name`	Specifies the name of the connection pool. If the name is the same as another existing OracleAS TopLink connection pool (such as the default OracleAS TopLink read pool), the existing connection pool is replaced with the new one. The `name` tag is mandatory.
`max-connections`	Specifies the maximum number of database connections that the connection pool can use. This tag is optional and accepts integer values. The default is 10.
`min-connections`	Specifies the minimum number of database connections that the connection pool should use at startup. This tag optional and accepts integer values. The default is 5.

Example 4-18 Configuring the Connection Pool Element

```
<session>
...
    <connection-pool>
        <is-read-connection-pool>true</is-read-connection-pool>
        <name>additionalReadPool</name>
        <max-connections>20</max-connections>
        <min-connections>10</min-connections>
        <login>
        ...
        </login>
    </connection-pool>
...
</session>
```

enable-logging Element

OracleAS TopLink does not automatically enable logging for a session unless you explicitly request it. To enable logging in a session, include the `enable-logging` element as part of your session definition in the `sessions.xml` file and set it to TRUE.

After you enable logging, you can customize the logging behavior on the session by including one or more logging options in the `sessions.xml` file. The available logging options appear in Table 4-8, and accept TRUE or FALSE as arguments.

Table 4-8 Logging Option Tags

Tag	Description
`log-debug`	Specifies whether the session logs debug information in addition to standard log entries.
`log-exceptions`	Specifies whether the session logs uncaught exception messages.
`log-exception-stacktrace`	Specifies whether the session logs exception stack traces.
`print-session`	Specifies whether the session logs session identifiers.
`print-thread`	Specifies whether the session logs thread identifiers.
`print-connection`	Specifies whether the session logs connection identifiers.
`print-date`	Specifies whether the session logs the date and time of each log entry.

Example 4-19 Configuring Logging and Logging Options

```
<session>
    ...
    <enable-logging>true</enable-logging>
    <logging-options>
        <log-debug>false</log-debug>
        <log-exceptions>true</log-exceptions>
```

```
        <log-exception-stacktrace>true</log-exception-stacktrace>
        <print-session>true</print-session>
        <print-thread>false</print-thread>
        <print-connection>true</print-connection>
        <print-date>true</print-date>
    </logging-options>
    ...
</session>
```

session-broker Element

The session broker enables client applications to view several databases through a single session. The `session-broker` element enables you to configure a session broker in the `sessions.xml` file, as follows:

1. Configure the session broker sessions in the `sessions.xml` file. These sessions are the database sessions or server sessions that the session broker uses to communicate with the databases.
2. Add the session broker to the `sessions.xml` file using the `session-broker` element.
3. Populate the `session-broker` element with a name and the sessions that you configured in Step 1.

Example 4-20 Configuring a Session Broker in the sessions.xml File

```
/* Configure the sessions for the SessionBroker */
<session>
    <name>EmployeeSession</name>
    ...
</session>
<session>
    <name>ProjectSession</name>
    ...
</session>
/* Configure the SessionBroker */
<session-broker>
/* Name the SessionBroker */
    <name>EmployeeAndProjectBroker</name>
/* Specify the sessions contained in the SessionBroker */
        <session-name>EmployeeSession</session-name>
        <session-name>ProjectSession</session-name>
</session-broker>
    ...
```

JTA Configuration

OracleAS TopLink J2EE integration includes support for JTA external connection pools and external transaction controllers. To enable a JTA external transaction controller, set the login to use an external transaction controller, and configure the following in your `sessions.xml` file:

- A JTA `DataSource` (in the `login` element)
- An external connection pool (in the `login` element)
- An external transaction controller (in the `session` element)

For more information about the OracleAS TopLink JTA integration, see "J2EE Integration".

Example 4-21 Configuring for JTA in the sessions.xml File

```
<session>
    ...
    <login>
        ...
        <uses-external-transaction-controller>true</uses-external-transaction-controller>
        <data-source>jdbc/MyApplicationDS</data-source>
        <uses-external-connection-pool>true</uses-external-connection-pool>
        ...
    </login>
    <external-transaction-controller-class>oracle.toplink.jts.oracle9i.Oracle9iJTSExternal
TransactionController</external-transaction-controller-class>
    ...
</session>
```

Example 4-22 Configuring for JTA in Code

```
DatabaseLogin login = null;
project = null;

/*note that useExternalConnectionPooling and useExternalTransactionController
must be set before Session is created */
project = new SomeProject();
login = project.getLogin();
login.useExternalConnectionPooling();
login.useExternalTransactionController();

/* usually, other login configuration such as user, password, JDBC URL comes
from the project but these can also be set here
session = new Session(project);

/* other session configuration, as necessary: logging, ETC
session.SetExternalTransactionController(new
SomeJTSExternalTransactionController());
session.login();
```

Registering Descriptors

How you add descriptors depends on how you created them. You can create project descriptors in the OracleAS TopLink Mapping Workbench and export them to a single descriptor file, set the `sessions.xml` file to reference the descriptor file. As a result, OracleAS TopLink can load the descriptors into the session automatically. A project class can also be specified in the `sessions.xml` file. For all other options, use the add descriptors method to register the descriptors, as Table 4-9, "addDescriptors Options" illustrates.

Table 4-9 addDescriptors Options

Format	Description
`addDescriptors(Project)`	Enables you to manually add additional descriptor to the session in the form of a project.
`addDescriptors(Vector)`	Enables you to add a vector of individual descriptor files to the session in the form of a project.
`addDescriptor(Descriptor)`	Enables you to add individual descriptors to the session.

Registering Descriptors after Login

You can register descriptors after the session logs in. Doing this enables you to load self-contained sub-systems after the session connects. Descriptors that are registered this way are independent of descriptors that are already registered.

- To change a descriptor and redeploy it with a minimum of down time, you can also re-register descriptors that are loaded in the session. You must also re-register all related descriptors at the same time, because changes to one descriptor may affect the initialization of other descriptors.

Caching Objects

Database sessions include an identity map that maintains object identity, and acts as a cache. When the session reads objects from the database, it instantiates them and stores them in the identity map. When the application subsequently queries for the same object, OracleAS TopLink returns the object in the cache rather than read the object from the database again.

You can force OracleAS TopLink to flush all objects from the cache. To do so, first ensure that none of the objects are in use within the database session. Then call the `initializeIdentityMaps()` method.

To improve performance, you can customize the identity map. For more information about using the identity map and caching, see the *Oracle Application Server TopLink Mapping Workbench User's Guide*.

Session Manager

The OracleAS TopLink session manager enables developers to build a series of sessions that are maintained under a single entity. The session manager is a static utility class that loads OracleAS TopLink sessions from the `sessions.xml` file, caches the sessions by name in memory, and provides a single access point for OracleAS TopLink sessions.

The session manager supports the following session types:

- `ServerSession` (see "Server Session and Client Session")
- `DatabaseSession` (see "Database Session")
- `SessionBroker` (see "Session Broker")

The session manager has two main functions: it creates instances of these sessions and it ensures that only a single instance of each named session exists for any instance of a session manager.

Instantiate the session manager as follows:

```
SessionManager.getManager()
```

This section describes techniques for working with the session manager and includes discussions of the following topics:

- Retrieving a Session from a Session Manager
- Storing Sessions in the Session Manager Instance

Retrieving a Session from a Session Manager

OracleAS TopLink maintains only one instance of the session manager class. The singleton session manager maintains all the named OracleAS TopLink sessions at runtime. When an application requests a session by name, the session manager retrieves the specified session from the configuration file.

To access the session manager instance, invoke the static `getManager()` method on the `oracle.toplink.tools.sessionmanagement.SessionManager` class. You can then use the session manager instance to load OracleAS TopLink sessions.

Example 4-23 Loading a Session Manager Instance

```
import oracle.toplink.tools.sessionmanagement.SessionManager;
SessionManager sessionManager = SessionManager.getManager();
```

OracleAS TopLink uses a class loader to load the session manager. The session manager, in turn, uses that same class loader to load named sessions that are not already initialized in the session manager cache.

Note:

To fully leverage the methods associated with the session type that is being instantiated, cast the session that is returned from the `getSession()` method. This type must match the session type that is defined in the `sessions.xml` file for the named session.

Example 4-24 Loading a Named Session from Session Manager Using Defaults

```
/* This example loads a named session (mysession) defined in the sessions.xml
file. */
SessionManager manager = SessionManager.getManager();
Server server = (Server) manager.getSession("myserversession");
```

Loading a Session with an Alternative Class Loader

You can use an alternative class loader to load sessions. This is common when your OracleAS TopLink application integrates with a J2EE container. If the session is not already in the session manager's in-memory cache of sessions, the session manager creates the session and logs in.

Example 4-25 Loading a Session Using an Alternative Class Loader

```
/* This example uses the specified ClassLoader to load a session (mysession)
defined in the sessions.xml file. */
ClassLoader classLoader = YourApplicationClass.getClassLoader();
SessionManager manager = SessionManager.getManager();
Session session = manager.getSession("mysession", // session nameclassLoader);
// classloader
```

Loading an Alternative Session Configuration File

You can use the XML Loader to load any XML configuration file on the application class path. This enables you to use files other than the standard `sessions.xml` file to load sessions.

You can use the XML loader to load different sessions, and even different class loaders, from configuration files. The `XMLLoader` class defines two constructors:

- The *zero-argument* constructor loads the default `sessions.xml` file.
- The single argument constructor includes a parameter (a `String`) that specifies an alternative configuration file.

Example 4-26 Loading an Alternative Configuration File

```
/* XMLLoader loads the toplink-sessions.xml file */
XMLLoader xmlLoader = new XMLLoader("toplink-sessions.xml");
ClassLoader classLoader = YourApplicationClass.getClassLoader();
SessionManager manager = SessionManager.getManager();
Session session = manager.getSession(
    xmlLoader, // XML Loader
    "mysession", // session name
    classLoader); // classloader
```

Reusing the Configuration File

If your application maintains the XML loader instance, then OracleAS TopLink reads sessions from the configuration file with the first `getsession ()`, but does not reparse the file with each subsequent `getsession ()` calls. If OracleAS TopLink uses a different XML loader to call a session, or if you invoke the API to refresh the configuration file, then OracleAS TopLink reparses the configuration file, but sessions already in the session manager do not change.

Opening Sessions without Logging In

The XML loader enables you to call a session using `getSession()`, without invoking the `login()` method. This enables you to prepare a session for use and leave login to the application.

Example 4-27 Open Session with No Login

```
SessionManager manager = SessionManager.getManager();
Session session = manager.getSession(
    new XMLLoader(), // XML Loader (sessions.xml file)
    "mysession", // session name
    YourApplicationClass.getClassLoader(), // classloader
    false, // log in session
    false); // refresh session
```

Reparsing the Session Configuration File

The XML loader can force OracleAS TopLink to reparse the session configuration file for sessions that do not exist in its in-memory cache. This function is useful when you want to add a session to an in-production `sessions.xml` file that already exists in the session manager cache. When the session manager attempts to load a session that is not in its in-memory cache, it reparses the XML file.

Example 4-28 Forcing a Reparse of the sessions.xml File

```
//In this example, the XML loader loads the sessions.xml file from the class
path.
SessionManager manager = SessionManager.getManager();
Session session = manager.getSession(
    new XMLLoader(), // XML Loader (sessions.xml file)
    "mysession", // session name
    YourApplicationClass.getClassLoader(), // classloader
    true, // log in session
    true); // refresh session
```

Storing Sessions in the Session Manager Instance

You can manually create a session in your application, rather than loading a preconfigured session from the session configuration file. Use the `SessionManager` class as a singleton to store the manually created session. Use the `getSession()` API with the single `String [session name]` argument on session manager to load the session.

Note:

The `getSession()` API is not necessary if you are loading sessions from a session configuration file.

Example 4-29 Storing Sessions Manually in the Session Manager

```
// create and log in session programmatically
Session theSession = project.createDatabaseSession();
theSession.login();
// store the session in the SessionManager instance
SessionManager manager = SessionManager.getManager();
manager.addSession("mysession", theSession);
// retrieve the session
Session session = SessionManager.getManager().getSession("mysession");
```

Destroying Sessions in the Session Manager Instance

The Session Manager provides two utility methods for destroying stored sessions.

Example 4-30 Destroying Sessions in the Session Manager

```
// create and log in session programmatically
Session theSession = project.createDatabaseSession();
theSession.login();
// store the session in the SessionManager instance
SessionManager manager = SessionManager.getManager();
manager.addSession("mysession", theSession);
...
// destroying the session
// this will throw a validation exception if the session name
// is not found
manager.destroySession("mySession");

OR

// if multiple sessions have been stored and all need to be
// destroyed, then use the destroyAllSessions API
manager.destroyAllSessions();
```

Session Querying

The Session class and its subclasses provide *query methods* that enable you to run queries against the object model rather than the relational model. You can invoke query methods using any of the following:

- Simple Query API
- Query Objects
- Predefined Queries

This section introduces query methods.

For more in-depth information, see "Session Queries".

Simple Query API

The Session class offers the following methods to access the database:

- The `readObject()` method uses a primary key to search for a single object in the database or the session cache. Specify the class of the queried object.

  For example:

  ```
  session.readObject(MyDomainObject.class);
  ```

  This example returns the first instance of `MyDomainObject` found in the table that contains the `MyDomainObject` class. If the query does not find an object that matches the criteria, it returns null. For more complex `readObject()` queries, augment the query with an OracleAS TopLink Expression.

  For more information, see "Using Expressions in Session Queries".

- The `readAllObjects()` method retrieves a `Vector` of objects from the database. Specify the class of the queried object.

  For example:

  ```
  session.readAllObjects(MyDomainObject.class)
  ```

  If the query does not find any objects that match the criteria, it returns an empty vector. For more complex `readAllObjects()` queries, augment the query with an OracleAS TopLink Expression.

  For more information, see "Using Expressions in Session Queries".

  The `readAllObjects()` method does not order the objects, but instead returns objects in the order in which they are found.

Using Expressions in Session Queries

To form more complex queries, include expressions in session query methods. Expression support makes up two public classes:

- The *Expression* class enables you to build either simple or complex logic into the expression, You can also combine multiple expressions in a query method.
- The *ExpressionBuilder* class is the factory that constructs new expressions.

To combine expressions with query methods, use the Expression Builder to create an expression and set the expressions as the selection criterion for the query.

Example 4-31 The readObject() Method Using an Expression

```
Employee employee = (Employee) session.readObject(Employee.class, new
ExpressionBuilder().get("lastName").equal("Smith"));
```

Example 4-32 The readAllObjects() Method Using an Expression

```
Vector employees = session.readAllObjects(Employee.class,new
ExpressionBuilder.get("salary").greaterThan(10000));
```

For more information about the OracleAS TopLink Expression Builder, see "Expressions".

Custom SQL Queries

You can execute custom SQL queries and stored procedure calls from within an OracleAS TopLink application. This is useful when you call stored procedures on the database and to access raw data. Use custom SQL strings and stored procedure calls in either of the following ways:

- Use the `executeSelectingCall()` and `executeNonSelectingCall()` session methods to execute SQL queries directly on the database.

  For example:

  ```
  Vector rows = session.executeSelectingCall(new SQLCall("SELECT USER, SYSDATE
  FROM DUAL"));
  ```

- Call the `executeQuery()` method on the `DatabaseSession`. The following code example uses SQL to read all employee IDs:

  ```
  DirectReadQuery query = new DirectReadQuery();
  query.setSQLString("SELECT EMP_ID FROM EMPLOYEE");
  Vector ids = (Vector) session.executeQuery(query);
  ```

Session Methods and the Unit of Work

If you call a session method to execute native SQL or invoke a stored procedure within a Unit of Work, then the Unit of Work is aware that you called a session method. However, it does not know about any changes the SQL or stored procedure makes to the database outside of the Unit of Work context, and so cannot roll back those changes if the `commit` call fails. Avoid using session methods inside a Unit of Work.

Query Objects

A query object is an OracleAS TopLink querying mechanism that offers full database querying access. Query objects support search criteria specified in several ways, including OracleAS TopLink expressions.

Use query objects to perform complex querying. An application creates query objects by instantiating the object and defining its querying criteria with either `Expression` objects or SQL strings.

You can:

- Execute the query objects directly, by calling the `executeQuery()` method on the `DatabaseSession`.
- Define new querying routines and add the routines to the session. Because you name these queries when you add them to the session, you can call them by name.

- Change the default querying behavior for read or write operations. An application can customize how the session's queries operate by supplying query objects to the descriptor's query manager.
- Change the default querying behavior for complex relationship mappings such as selection queries.

For more information about creating and using query objects, see "Query Objects".

Predefined Queries

Predefined queries are queries you store in either the `sessions.xml` file or the OracleAS TopLink descriptor. Because they are part of the session or descriptor, OracleAS TopLink stores predefined queries in memory after you initially invoke them. Use predefined queries to maintain frequently-called queries.

For more information about predefined queries, see "Predefined Queries".

Session Types

OracleAS TopLink provides several session types that enable you to tailor the session to your application needs. This section describes the following OracleAS TopLink session types:

- Server Session and Client Session
- Database Session
- Session Broker
- Remote Session

Server Session and Client Session

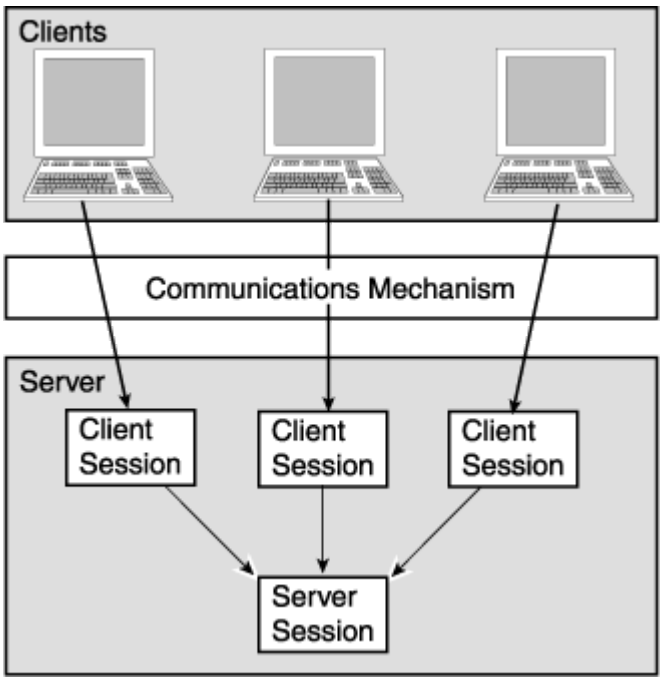
The server session and client session architecture is known collectively as a *three-tier* architecture. In this type of architecture, the server session provides session management for the clients, and the client session acts as an dedicated database session for each client or request.

Although they are two separate session types, use the client sessions and server sessions together. You define the server session in the `sessions.xml` file. After you instantiate the server session, you acquire client sessions from it. Each client session can have only one associated server session, but a server session can support any number of client sessions.

Three-Tier Architecture Overview

In an OracleAS TopLink three-tier architecture, client sessions and server sessions both reside on the server. Client applications access the OracleAS TopLink application through a client session, and the client session communicates with the database using the server session.

Figure 4-6 Server Session and Client Session Usage



Text description of the illustration cliserv.gif

EJBs and Server Session

The Enterprise JavaBean (EJB) container manages interaction with the database and OracleAS TopLink. The server session manages all aspects of persistence, such as caching, reading and writing, but does so behind the scenes.

General Concepts for the OracleAS TopLink Three-Tier Design

Although the server session and the client session are two different session types, you can treat them as a single unit in most cases, because they are both required to provide three-tier functionality to the application. The server session provides the client session to client applications, and also supplies the bulk of the session functionality. This section discusses some of the advantages and general concepts associated with the OracleAS TopLink three-tier design.

Shared Resources

The three-tier design enables multiple clients to share persistent resources. The server session provides its client sessions with a shared live object cache, read and write connection pooling, and parameterized named queries. Client sessions also share descriptor metadata.

You can use client sessions and server sessions in any application server architecture that allows for shared memory and supports multiple clients. These architectures can include HTML, Servlet, JSP, RMI, CORBA, DCOM, and EJB.

To support a shared object cache, client sessions must:

- Implement any changes to the database with the OracleAS TopLink Unit of Work.
- Share a common database login for reading (you can implement separate logins for writing).

For more information, see "Sessions and the Cache".

Providing Read Access

To read objects from the database the client must first acquire a client session from the server session. Acquiring a client session gives the client access to the session cache and the database through the server session.

Example 4-33 Acquiring a Client Session

```
ClientSession myClientSession = myServerSession.acquireClientSession();
```

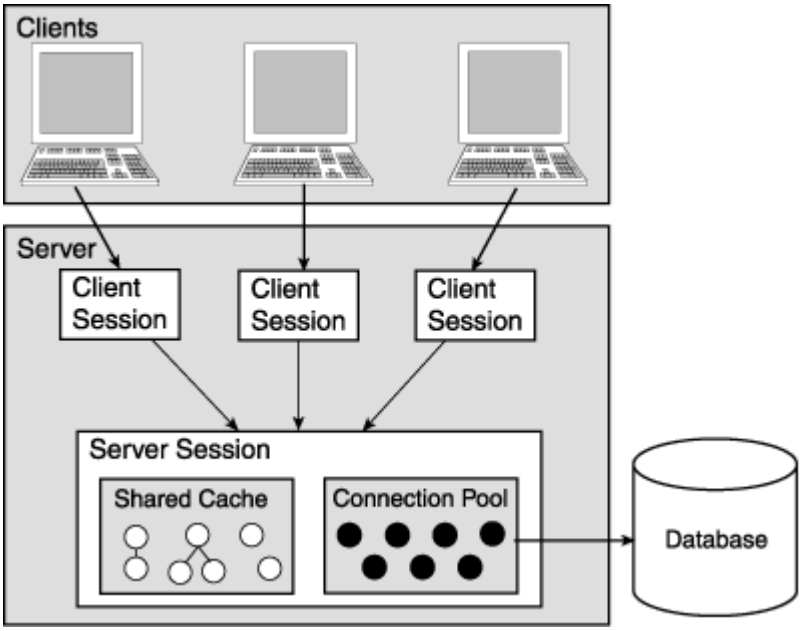
After the client acquires a client session, it can send read requests to the server. The server session responds to these requests as follows:

- If the object or data is in the session cache, then the server session returns the information back to the client.
- If the object or data is not in the cache, then the server session reads the information from the database and stores the object in the session cache. The objects are then available for retrieval from the cache.

Because a server session processes each client request in a separate thread, this enables multiple clients to access the database connection pool concurrently.

Figure 4-7 illustrates how multiple clients read from the database using the server session.

Figure 4-7 Multiple Client Sessions Reading the Database Using the Server Session



Text description of the illustration mltiread.gif

To read objects from the database using a Client Session:

1. Start the application server.
2. Create a `ServerSession` object and call `login()`.
3. Call `acquireClientSession()` to acquire a `ClientSession` from the `ServerSession`.

4. Execute read operations on the `ClientSession` object.

Note:

Do not use the `ServerSession` object directly to read objects from the database.

Providing Write Access

Because the client session disables all database modification methods, a client session cannot create, change, or delete objects directly. Instead, the client must obtain a Unit of Work from the client session to perform database modification methods.

To write to the database, the client acquires a client session from the server session and then acquires a `UnitOfWork` within that client session. The Unit of Work acts as an exclusive transactional object space, and also ensures that any changes that are committed to the database also occur in the session cache.

Caution:

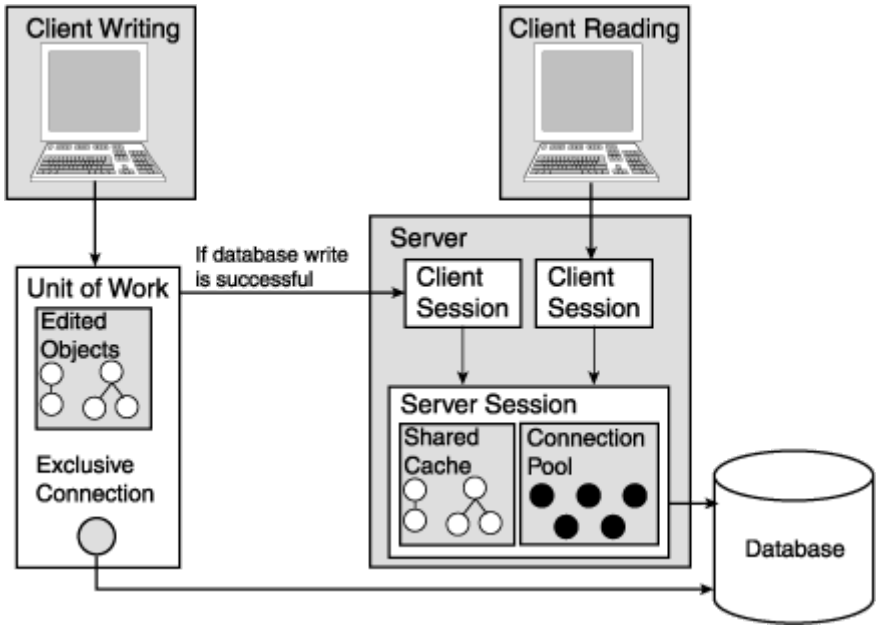
Although client sessions are thread-safe, do not use them to write across multiple threads. Multi-thread writes from the same client session can result in errors and a loss of data.

To write to the database using a Unit of Work:

1. Start the application server.
2. Create a `ServerSession` object and call `login()`.
3. Call `acquireClientSession()` to acquire a `ClientSession` from the `ServerSession`.
4. Acquire a `UnitOfWork` object from the `ClientSession` object.

   For more information about the Unit of Work, see Chapter 7, "Transactions".
5. Perform the required updates, and then commit the `UnitOfWork`.

Figure 4-8 Writing with Client Sessions and Server Sessions



Text description of the illustration uowwrite.gif

Parallel Units of Work

The Unit of Work ensures that the client edits objects in a separate object transaction space. This feature enables clients to perform object transactions in parallel. When transactions commit, the Unit of Work makes any required changes in the database and then merges the changes into the shared OracleAS TopLink session cache. The modified objects are then available to all other users.

For more information about the Unit of Work, see to Chapter 7, "Transactions".

Security and User Privileges

You can define several different server sessions in your application to support users with different data access rights. For example, your application may serve a group called "Managers," who has access rights to salary information, and a group called "Employees," who do not. Because each session you define in the `sessions.xml` file has its own login information, you can create multiple sessions, each with its own login credentials, to meet the needs of both of these groups.

Concurrency

The server session supports concurrent clients by providing each client with a dedicated thread of execution. Dedicated threads enable clients to operate asynchronously--that is, client processes execute as they are called and do not wait for other client processes to complete.

OracleAS TopLink safeguards thread safety with a concurrency manager. The concurrency manager ensures that no two threads interfere with each other when performing operations such as creating new objects, executing a transaction on the database, or accessing valueholders.

Not all JDBC drivers support concurrency. Those that do not may require a thread to have exclusive access to a JDBC connection when reading. Configure the server session to use exclusive read connection pooling in these cases.

Connection Pooling

When you instantiate the server session, it creates a pool of database connections. It then manages the connection pool based on your session configuration, and shares the connections among its client sessions. The server session provides connections to client sessions on an as-needed basis. When the client session releases the connection, the server session recovers the connection and makes it available to other client processes. Reusing connections reduces the number of connections required by the application and allows a server session to support a larger number of clients.

By default, the OracleAS TopLink write connection pool maintains a minimum of five connections and a maximum of ten. You can change these settings as follows:

- *To change the settings for the entire project*, adjust these settings in the OracleAS TopLink Mapping Workbench.

  For more information, see "Working with Connection Pools" in the *Oracle Application Server TopLink Mapping Workbench User's Guide*.

- *To change the settings for a particular server session*, adjust these settings in the `sessions.xml` file. You can make these changes using the OracleAS TopLink Sessions Editor, or add the following lines to the `session` element in the file manually:

```
<session>
...
    <connection-pool>
        ...
        <max-connections>20</max-connections>
        <min-connections>10</min-connections>
        ...
    </connection-pool>
...
</session>
```

Tip:

To maintain compatibility with JDBC drivers that do not support many connections, the default number of connections is small. If your JDBC driver supports it, use a larger number of connections for reading and writing.

The server session also supports multiple write connection pools and non-pooled connections. If your application server or JDBC driver also supports write connection pooling, you can configure the server session to use this feature. Set these options at the session level, modify the `session` element in the `sessions.xml` file.

For more information, see "Configuring Sessions with the sessions.xml File".

Read Connections

Although a single connection supports multiple threads reading asynchronously, some JDBC drivers perform better with multiple read connections. OracleAS TopLink enables you to allocate multiple read connections, and balances the load across the connections using a *least-busy* algorithm.

Server Session Connection Options

The server session maintains a pool of read connections and a pool of write connections for its client sessions. You can customize the following options either in the `sessions.xml` file or in Java code:

- Create a new connection pool and add it to the pools the server session

```
addConnectionPool(String poolName, JDBCLogin login, int
minNumberOfConnections, int maxNumberOfConnections)
```

- In Java code, configure the read connection pool:

```
useReadConnectionPool(int minNumberOfConnections, int
maxNumberOfConnections)
```

- In Java code, configure the read connection pool to allow only a single thread to access each connection:

```
useExclusiveReadConnectionPool(int minNumberOfConnections, int
maxNumberOfConnections)
```

- In Java code, set the maximum number of nonpooled connections:

```
setMaxNumberOfNonPooledConnections(int maxNumber)
```

Client Session Connection Options

The three ways to get connections from within a client session object correspond to three arguments you can pass with the `acquireClientSession()` method on the server session are:

- Pass no argument (the *zero argument*). The acquired `ClientSession` uses the default connection pool.
- Pass a `poolName` as an argument. The acquired `ClientSession` uses a connection from the specified pool.
- Pass a `DatabaseLogin` object as an argument. The acquired `ClientSession` uses a specified `DatabaseLogin` object to obtain a connection.

By default, the server session does not allocate database connections for these client session until a Unit of Work commits to the database (a *lazy* database connection).

If you need to establish database connection immediately, configure the `ConnectionPolicy` object to specify a connection option more suited to your needs, and pass the `ConnectionPolicy` object as an argument.

Connection Policy

The `ConnectionPolicy` class provides the following methods to configure a client connection:

- `setPoolName(String poolName)`: Creates a connection from the named connection pool. You can also use the `ConnectionPolicy(String poolName)` method.
- `setLogin(DatabaseLogin login)`: Sets up a connection by logging directly into the database. You can also use the `ConnectionPolicy(DatabaseLogin login)` method from the connection policy constructor.

- `useLazyConnection()`: Specifies whether the application uses a lazy connection (a connection that OracleAS TopLink instantiates only when required).
- `setLazyConnection(boolean isLazy)`: Specifies a lazy connection.
- `dontUseLazyConnection()`: Creates an active connection.

If you request a database connection when none is available, the method waits for the next available connection, rather than time out or return an error.

Reference

Table 4-10 and Table 4-11 summarize the most common public methods for `ClientSession` and `ServerSession`. For more information about the available methods for `ClientSession` and `ServerSession`, see the *Oracle Application Server TopLink API Reference*.

Table 4-10 Elements for Client Session

Element	Method Name
Executing a query object	`executeQuery(DatabaseQuery query, Vector parameters)`
Reading from the database	`readAllObjects(Class domainClass, Expression expression)` `readObject(Class domainClass, Expression expression)`
Release	`release()`
Unit of Work	`acquireUnitOfWork()`

Table 4-11 Elements for Server Session

Element	Method Name
Acquire `ClientSessions`	`acquireClientSession()`
Logging (Logging is not turned on, by default)	`logMessages()`
Login / logout	`login()` `logout()`

Customizing Server Session and Database Login

You can use a session amendment class to configure the server session and database login in ways not available through the deployment descriptor file. For example, you can:

- Specify special settings for the JDBC driver. For example, if you are working with an incompatible database driver, you can implement *parameter binding*, to enable a different data conversion routine.

- Access regular OracleAS TopLink features, such as database connections or caching, directly.
- Define custom finder queries on one or more OracleAS TopLink descriptors (under EJB 1.1).
- Enable native SQL support if your JDBC bridge does not support the JDBC standard SQL syntax.
- Enable binding and parameterized SQL, to specify whether values are inlined directly into the generated SQL or are parameterized.
- Enable batch writing, forcing the application to send groups of insert, update, and delete statements to the database in a single batch.
- Optimize data conversion.

Working with Login

Databases generally require a valid user name and password to login successfully. OracleAS TopLink applications maintain this information in the `DatabaseLogin` class. All sessions must have a valid `DatabaseLogin` instance before logging in to the database.

For more information about the `DatabaseLogin`, see "Database Session".

Registering Event Listeners for EJB 1.1

To customize an EJB 1.1 application, register a session listener class that extends `oracle.toplink.sessions.SessionEventAdaptor`. Configure the listener to listen for various session events, such as `pre_login` and `post_commit_unit_of_work`. To register the OracleAS TopLink session, define the `event_listener_class` tag in the `toplink-ejb-jar.xml` file, as follows:

```
<session>
    <event_listener_class>
    oracle.toplink.ejb.cmp.demos.sessionlistener
    </event_listener_class>
</session>
```

Specify the fully-qualified name of the class that you want to use for this purpose in the `customization-class` element of the `toplink-ejb-jar.xml` deployment descriptor.

Example 4-34 illustrates the project portion of `the toplink-ejb-jar.xml` deployment descriptor that specifies a `customization class`.

Example 4-34 Customization Class in the toplink-ejb-jar.xml File

```
<session>
    <name>EmployeeDemo</name>
    <project-class>
    oracle.toplink.demos.ejb.cmp.wls.employee.EmployeeProject.class
    </project-class>
    <login>
        <connection-pool>ejbPool</connection-pool>
    </login>
    <customization-class>
    oracle.toplink.demos.ejb.cmp.wls.employee.EmployeeCustomizer
    </customization-class>
</session>
```

Database Session

A database session is the simplest session OracleAS TopLink offers. The database session offers functionality for a single user and a single database connection.

Note:

Use server sessions and client sessions for three-tier applications; applications that are built using database sessions may be difficult to migrate to a scalable architecture in the future.

A database session contains and manages the following information:

- An instance of `Project` and `DatabaseLogin`, which stores database login and configuration information
- The JDBC connection and the database access
- The descriptors for each of the application persistent classes
- Identity maps that maintain object identity and act as a cache

Creating a Database Session

An application opens a database session by creating an instance of the `DatabaseSession` class, and initializing the project with the appropriate database login parameters. After initialization, the session:

- Registers the OracleAS TopLink descriptors (see "Registering Descriptors")
- Connects to the database
- Establishes the session cache

Connecting to the Database

After you register the descriptors, use the `DatabaseSession` class to connect to the database, using the `login()` method. If the login parameters in the `DatabaseLogin` class are incorrect, or if the connection cannot be established, OracleAS TopLink throws a `DatabaseException`.

After a connection is established, the application can use the session to access the database. To test the connection, invoke the `isConnected()` method. If the connection is functions, that method returns TRUE.

To interact with the database, the application use the session querying methods or executes query objects. The interactions between the application and the database are collectively known as the query framework. For more information about querying, see Chapter 6, "Queries".

Although session query methods work well with database sessions, concurrency issues make the database session unsuited for three-tier applications.

Logging Out of the Database

To log out the session, use the `logout()` method. To disconnect the session from the relational database and flush the session's identity maps, call the `logout()` method.

Because logging in to the database can be time consuming, log out only when all database interactions are complete.

Applications that log out from the database do not have to reregister their descriptors if they log back in to the database.

Using Manual Transaction Control

Certain versions of Sybase JConnect prevent the execution of stored procedures with JDBC auto-commit. If you use OracleAS TopLink with a version of JConnect that causes this problem, use the `handleTransactionsManuallyForSybaseJConnect()` method to handle the transactions manually.

To add transaction processing to a set of database operations:

1. At the start of the transaction set, call `beginTransaction()`.
2. Specify a try-catch block that calls `rollbackTransaction()` if a database exception is thrown.
3. At the end of the transaction set, call `commitTransaction()`.

Note:

The Unit of Work is already transaction bound and does not require these calls.

Example 4-35 A Typical Manual Transaction

```
/** Update a group of employee records*/
void writeEmployees(Vector employees, Session session)
{
    Employee employee;
    Enumeration employeeEnumeration = employees.elements();
    try {
        session.beginTransaction();
        while (employeeEnumeration.hasMoreElements())
        {
            employee=(Employee) employeeEnumeration.nextElement();
            session.writeObject(employee);
        }
        session.commitTransaction();
    } catch (DatabaseException exception) {
    // If a database exception has been thrown, roll back the transaction.
        session.rollbackTransaction();
    }
}
```

Creating Database Sessions: Examples

Example 4-36 Creating a Session from a OracleAS TopLink Mapping Workbench Project

```
import oracle.toplink.tools.workbench.*;
import oracle.toplink.sessions.*

// Create the project object
Project project = XMLProjectReader.read("C:\TopLink\example.xml");
DatabaseLogin loginInfo = project.getLogin();
loginInfo.setUserName("scott");
```

```
loginInfo.setPassword("tiger");

//Create a new instance of the session and login
DatabaseSession session = project.createDatabaseSession();
try {
    session.login();
    } catch (DatabaseException exception) {
    throw new RuntimeException("Database error occurred at login: " +
    exception.getMessage());
    System.out.println("Login failed");
    }

/* Do any database interaction using the query framework, transactions or units
of work */
...

// Log out when database interaction is over
session.logout();
Creating and using a session from coded descriptors
import oracle.toplink.sessions.*;

//Create the project object.
DatabaseLogin loginInfo = new DatabaseLogin();
loginInfo.useJDBCODBCBridge();
loginInfo.useSQLServer();
loginInfo.setDataSourceName("MS SQL Server");
loginInfo.setUserName("scott");
loginInfo.setPassword("tiger");
Project project = new Project(loginInfo);

//Create a new instance of the session, register the descriptors, and login
DatabaseSession session = project.createDatabaseSession();
session.addDescriptors(this.buildAllDescriptors());
try {
    session.login();
} catch (DatabaseException exception) {
    throw new RuntimeException("Database error occurred at login: " +
    exeption.getMessage());
    System.out.println("Login failed");
    }

//Do any database interaction using the query framework, transactions or units
of work
...
//Log out when database interaction is over
session.logout();
}
```

Reference

Table 4-12 summarizes the most common public methods for the `DatabaseSession` class. For more information about the available methods for the `DatabaseSession` class, see the *Oracle Application Server TopLink API Reference*.

Table 4-12 Elements for Database Session

Description	Method Name
Construction methods	`Project.createDatabaseSession()`
log in to the database (Defaults to the `user name` and `password` from project login)	`login()`
Log out of the database	`logout()`

Description	Method Name
Executing predefined queries	`executeQuery(String queryName)`
Executing a query object	`executeQuery(DatabaseQuery query)`
Reading from the database	`readAllObjects(Class domainClass, Expression expression)` `readObject(Class domainClass, Expression expression)`
SQL logging (logging is off by default)	`logMessages()`
Debugging	`printIdentityMaps()`

Transactions	`beginTransaction()`
	`commitTransaction()`
	`rollbackTransaction()`

Exception handlers (throws exceptions by default)	`setExceptionHandler(ExceptionHandler handler)`
JTA/JTS (Defaults to use JDBC transactions)	`setExternalTransactionController(ExternalTransactionController controller)`
Unit of Work	`acquireUnitOfWork()`
Writing to the database	`deleteObject(Object domainObject)` `writeObject(Object domainObject)`

Session Broker

OracleAS TopLink provides the session broker to enable multiple database access. Use the session broker to access objects that are stored on multiple databases. The session broker:

- Provides transparent multiple database access through a single OracleAS TopLink session
- Enables objects to reference objects on other databases
- Transparently manages new object storage in a multiple databases environment
- Manages single Unit of Work and transaction across multiple databases
- Supports two-phase commit when integrated with a compliant JTA driver; otherwise, uses a two stage algorithm

Multiple Sessions

The session broker is a powerful tool that enables you to use data that is split across multiple databases for a given application. An alternative to the session broker is to use multiple

sessions to work with multiple databases:

- If the data on each database is unrelated to data on the other databases, and relationships do not cross database boundaries, then you can create a separate session for each database. For example, you might have individual databases and associated sessions dedicated to each cost center.

  This arrangement requires that you to manage each session manually and ensure that the class descriptors for your project reside in the correct session.

- You can use additional sessions to house a regular batch job. In this case, you can create two or more sessions on the same database. In addition to the main session that supports client queries, you create other sessions that support batch inserts at low-traffic times in your system. This enables you to maintain the client cache.

Configuring the Session Broker in Code

After the session broker is set up and logged in, it functions like a session, making multiple database access transparent. Because a session broker is more complex than a regular database session, it is requires more work to create and configure.

Configuring the Session Broker in the Sessions.xml file

To configure the session broker in the `sessions.xml` file, configure sessions for use in the session broker, and then reference the sessions from within the `session-broker` element. When the session manager instantiates the session broker, it also instantiates the referenced sessions.

For more information, see "session-broker Element".

Configuring Session Broker in Java Code

Because the session broker references other sessions, configure these sessions before instantiating the session broker. Add all required descriptors to the session, but do not initialize the descriptors, or log the sessions in. The session broker manages these issues when you instantiate it.

After you configure a session, use the `registerSession(String name, Session session)` method to register it with a `SessionBroker`.

Example 4-37 Adding Sessions to a Session Broker

This code prepares and adds two sessions to a session broker.

```
Project p1 = ProjectReader.read(("C:\Test\Test1.project"));
Project p2 = ProjectReader.read(("C:\Test\Test2.project"));

/* modify the user name and password if they are not correct in the
.project file */
p1.getLogin().setUserName("User1");
p1.getLogin().setPassword("password1");
p2.getLogin().setUserName("User2");
p2.getLogin().setPassword("password2");
DatabaseSession session1 = p1.createDatabaseSession();
DatabaseSession session2 = p2.createDatabaseSession();

SessionBroker broker = new SessionBroker();
broker.registerSession("broker1", session1);
broker.registerSession("broker2", session2);
```

```
broker.login();
```

When you call the `login()` method on the session broker, session broker logs in all contained sessions and initializes the descriptors in the sessions. After login, the session broker appears and functions as a regular session. OracleAS TopLink handles the multiple database access transparently.

Example 4-38 Writing to the Database

```
UnitOfWork uow = broker.acquireUnitOfWork();
Test test = (Test) broker.readObject(Test.class);
Test testClone = uow.registerObject(test);
. . .
//change and manipulate the clone and any of its references
. . .
uow.commit();

//log out when finished
broker.logout();
```

Committing a Transaction with a Session Broker

If you use a session broker, incorporate a JTA external transaction controller wherever possible. The external transaction controller provides a two-phase commit, which passes the SQL statements that are required to commit the transaction to the JTA driver. The JTA driver handles the entire commit process.

JTA guarantees that the transaction commits or rolls back completely, even if the transaction involves more than one database. If the commit to any one database fails, then all database transactions roll back. The two-phase commit is the safest method available to commit a transaction to the database.

Two-phase commit support requires integration with a compliant JTA driver.

For more information about the JTA drivers, see "JTA".

Committing a Session without a JTA Driver: Two-stage Commits

If there is no JTA driver available, then the session broker provides a *two-stage commit* algorithm. A two-stage commit differs from a two-phase commit in that it only guarantees data integrity up to the point of the final commit of the transaction. If the SQL executes successfully on all databases, but the commit then fails on one database, only the database that experiences the commit failure rolls back.

Although unlikely, this scenario is possible. As a result, if your system does not include a JTA driver and you use a two-stage commit, build a mechanism into your application to deal with this type of potential problem.

Using the Session Broker in a Three-tier Architecture

The session broker operates in a seamless manner in a tree-tier environment. To use session broker in a three-tier application, configure server sessions for the session broker.

Although you can configure your session broker in code as illustrated in Example 4-39, we recommend you use the OracleAS TopLink Sessions Editor to specify a session broker in the `sessions.xml` file.

For more information, see the *Oracle Application Server TopLink Mapping Workbench User's Guide*.

Example 4-39 Configuring a Session Broker in a Three-Tier Architecture in Java Code

```
Project p1 = ProjectReader.read(("C:\Test\Test1.project"))

Project p2 =
ProjectReader.read(("C:\Test\Test2.project"));

/* Create Sessions for the SessionBroker */
Server sSession1 = p1.createServerSession();
Server sSession2 = p2.createServerSession();


/* Create the SessionBroker and assign the sessions to it */
SessionBroker broker = new SessionBroker();
broker.registerSession("broker1", sSession1);
broker.registerSession("broker2", sSession2);
broker.login();
```

Clients with a Three-Tier Session Broker

When a three-tier session broker application uses server sessions to communicate with the database, clients require a client session to write to the database. Similarly, when you implement a session broker, the client requires a *client session broker* to write to the database.

A client session broker is a collection of client sessions, one from each server session associated with the session broker. When a client acquires a client session broker, the session broker collects one client session from each associated server session, and wraps the client sessions so that they appear as a single client session to the client application.

To request a client session broker, the client calls the `acquireClientSessionBroker()` method.

Example 4-40 Sample Client Request Code

```
Session clientBroker = broker.acquireClientSessionBroker();
```

Limitations

Using the session broker is not the same as linking databases at the database level. If your database allows linking, use that functionality to provide multiple database access.

The session broker has the following limitations:

- You cannot split multiple table descriptors across databases.
- Each class must reside on only one database.
- You cannot use joins through expressions across databases.
- Many-to-many join tables and direct collection tables must reside on the same database as the source object.

Note:

The "Advanced Use" section describes a work-around for this limitation. It uses an amendment to the descriptor.

Advanced Use

Many-to-many join tables and direct collection tables must be on the same database as the source object, because reading these tables requires a join that spans both databases. To get around this problem, use the `setSessionName(String sessionName)` method on `ManyToManyMapping` and `DirectCollectionMapping`. This method indicates that the join table or direct collection table is on the same database as the target table.

```
Descriptor desc = session1.getDescriptor(Employee.class);
((ManyToManyMapping)desc.getObjectBuilder().getMappingForAttributeName("projects
")).setSessionName("broker2");
```

`DatabaseQuery` offers a similar method that supports non object queries.

Reference

Table 4-13 summarizes the most common public methods for `SessionBroker`. For more information about the available methods for `SessionBroker`, see the *Oracle Application Server TopLink API Reference*.

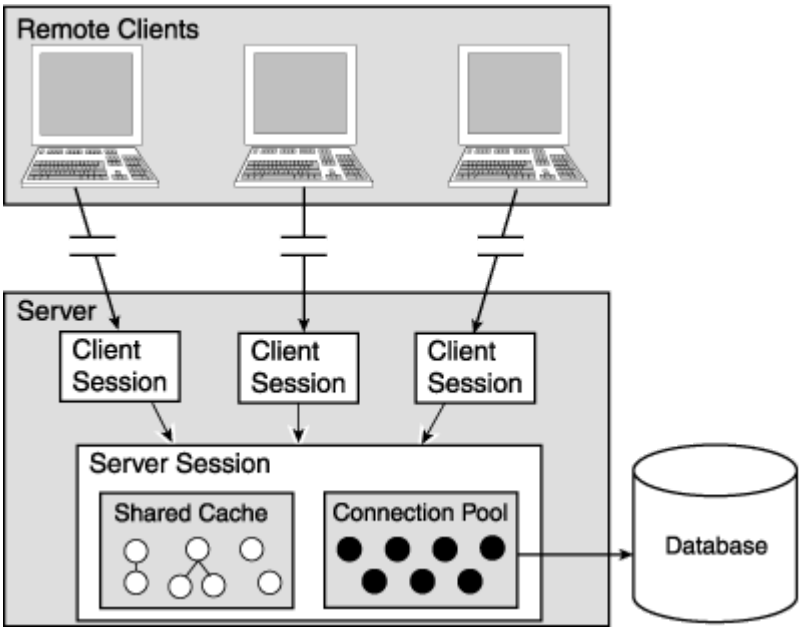
Table 4-13 Elements for the Session Broker

Element	Method Name
Writing objects	`acquireUnitOfWork()`
Acquiring `ClientSessions`	`acquireClientSessionBroker()`
Database connection	`login()` `logout()`

Remote Session

A remote session is a session that resides on the client. It communicates with a client session on the server, and the client session communicates with the server session on its behalf. Remote sessions handle object identity, proxies, and the communication between the client and server layer.

Figure 4-9 Remote Session Model for a Three-tier Application

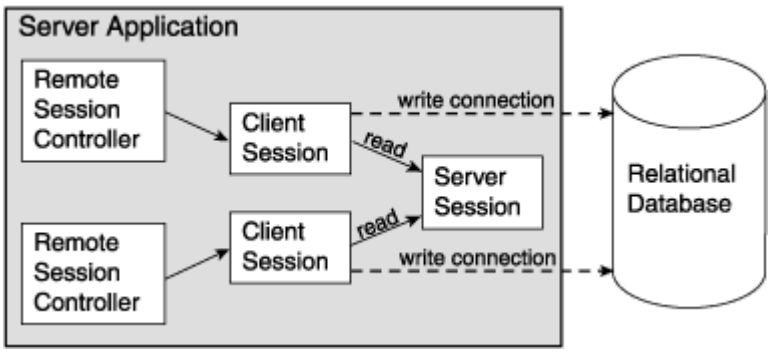


Text description of the illustration rmtiread.gif

The remote session can also interact with a database session rather than a client session. The user sets this up on the server side.

When choosing between a client session and a database session, you should be aware that the database session is not suited to a distributed environment, because the database session enables only one user to interact with the database. However, if the remote session interacts with a client session, then multiple remote sessions can share a single database connection. The remote session also benefits from connection pooling.

Figure 4-10 Remote Session and a Database or a Client Session



Text description of the illustration rmtdbses.gif

Architectural Overview

The remote session model consists of the following layers (also see Figure 4-11):

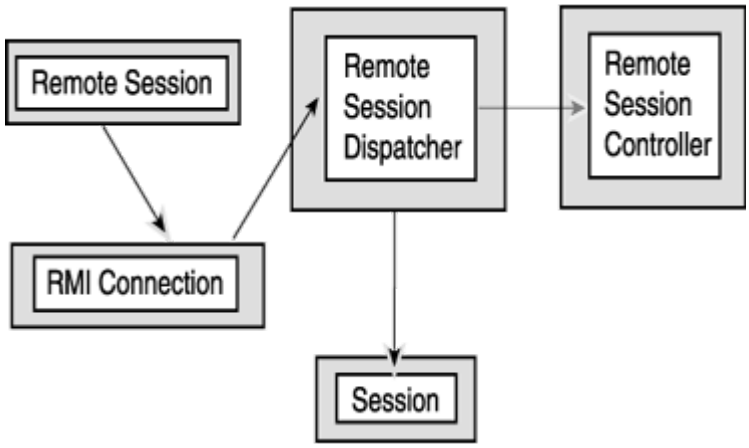
- The application layer--a client side application talking to remote session
- The transport layer--a communication layer, RMI or RMI-IIOP
- The server layer--OracleAS TopLink session communicating with a database

The request from the client application to the server travels down through the layers of a distributed system. A client that makes a request to the server session uses the remote session

as a conduit to the server session. The client references the remote session, and the remote session forwards a request to the server session through the transport layer.

At runtime, the remote session builds its knowledge base by reading descriptors and mappings from the server side as they are needed. These descriptors and mappings are lightweight because not all information is passed on to the remote session. The information needed to traverse an object tree and to extract primary keys from the given object is passed with the mappings and descriptors.

Figure 4-11 An Architectural Overview of the Remote Session



Text description of the illustration remarch.gif

Application layer

The application layer includes the application and the remote session. The remote session is a subclass of the session and maintains all the public protocols of the session, giving the appearance of working with the local database session.

The remote session maintains its own identity map and a hash table of all the descriptors read from the server. If the remote session can handle a request by itself, the request is not passed to the server. For example, a request for an object that is in the Remote session cache is processed by the remote session. However, if the object is not in the remote session cache, the request passes to the server session.

Transport Layer

The transport layer is responsible for carrying the semantics of the invocation. It is a layer that hides all the protocol dependencies from the application and server layer.

The transport layer includes a remote connection that is an abstract entity through which all requests to the server are forwarded. Each remote session maintains a single remote connection that marshals and unmarshals all requests and responses on the client side.

The remote session supports communications over RMI and CORBA. It includes deployment classes and stubs for RMI, BEA WebLogic RMI, VisiBroker, OrbixWeb, BEA WebLogic EJB, and Oracle 10i EJB.

Server Layer

The server layer includes a remote session controller dispatcher and a session. The remote session controller dispatcher marshals and unmarshals all responses and requests from the

server side. This is a client side component.

The remote session controller dispatcher is an interface between the session and transport layers. It hides the specifics of the transport layer from the session.

Securing Remote Session Access

The remote session represents a potential security risk because it requires you to register a remote session controller dispatcher as a service that anyone can access. This can expose the entire database.

To reduce this threat, run a server manager as a service to hold the remote controller session dispatcher. All the clients must then communicate through the server manager, which implements the security model for accessing the remote session controller dispatcher.

On the client side, the user requests the remote session controller dispatcher. The manager returns a remote session controller dispatcher only if the user has access rights according to the security model built into the server manager.

To access the system, the remote session controller dispatcher on the client side creates a remote connection, and acquires remote session from the remote connection. The API for the remote session is the same as for the session, and there is no user-visible difference between working on a session or a remote session.

Queries

Read queries are publicly available on the client side, but queries that modify objects must be performed using the Unit of Work.

Refreshing

Calling refresh methods on the remote session causes database reads, and may also cause cache updates if the data being refreshed is modified in the database. This can lead to poor performance.

To improve performance, configure refresh methods to run against the server session cache, by configuring the descriptor to always remotely refresh the objects in the cache on all queries. This technique ensures that all queries against the remote session refresh the objects from the server session cache, without the database access.

Cache hits on remote sessions still occur on read object queries based on the primary keys. If you want to avoid this, disable the remote session cache hits on read object queries based on the primary key.

Example 4-41 Refreshing on the Server Session Cache

```
// Get the PolicyHolder descriptor
Descriptor holderDescriptor = remoteSession.getDescriptor(PolicyHolder.class);

// Set refresh on the ServerSession cache
holderDescriptor.alwaysRefreshCachedOnRemote();

// Disable remote cache hits, ensure all queries go to the ServerSession cache
holderDescriptor.disableCacheHitsOnRemote();
```

Indirection

The remote session supports indirection objects. An indirection object is a valueholder that can be invoked remotely on the client side. When invoked, the valueholder first checks to see if the requested object exists on the remote session. If not, then the associated valueholder on the server is instantiated to get the value that is then passed back to the client. Remote valueholders are used automatically; the application's code does not change.

Cursored Streams

Remote session supports cursored streams, but not scrollable cursors.

For more information about enabling cursored streams, see "Java Streams".

Unit of Work

Use a Unit of Work acquired from the remote session to modify objects on the database. A Unit of Work acquired from the remote session offers the user the same functionality as a Unit of Work acquired from the client session or the database session.

Creating a Remote Connection Using RMIConnection

Example 4-42 and Example 4-43 demonstrate how to create a remote OracleAS TopLink session on a client that communicates with a remote session controller on a server that uses RMI. After creating the connection, the client application uses the remote session as it does with any other OracleAS TopLink session.

These examples assume that a class called `RMIServerManager` exists on the server. It is not an OracleAS TopLink-enabled class. This class has a method that instantiates and returns an `RMIRemoteSessionController` (an OracleAS TopLink server side interface).

Example 4-42 Client Acquiring RMIRemoteSessionController from Server

The client-side code gets a reference to the `RMIServerManager` and uses this code to get the `RMIRemoteSessionController` running on the server. The reference to the session controller is then used to create the `RMIConnection` from which it acquires a remote session.

```
RMIServerManager serverManager = null;
// Set the client security manager
try {
    System.setSecurityManager(new RMISecurityManager());
    } catch(Exception exception) {
    System.out.println("Security violation " + exception.toString());
}
// Get the remote factory object from the Registry
try {
    serverManager = (RMIServerManager) Naming.lookup("SERVER-MANAGER");
    } catch (Exception exception) {
    System.out.println("Lookup failed " + exception.toString());
}
// Start RMIRemoteSession on the server and create an RMIConnection
RMIConnection rmiConnection = null;
try {
    rmiConnection = new
    RMIConnection(serverManager.createRemoteSessionController());
    } catch (RemoteException exception) {
    System.out.println("Error in invocation " + exception.toString());
}
// Create a remote session which we can then use as a normal OracleAS TopLink
Session
Session session = rmiConnection.createRemoteSession();
```

Example 4-43 Server Creating RMIRemoteSessionController for Client

The `RMIServerManager` uses this code to create and return an instance of an `RMIRemoteSessionController` to the client. The controller sits between the remote client and the local OracleAS TopLink session.

```
RMIRemoteSessionController controller = null;
try {
    /* Create instance of RMIRemoteSessionControllerDispatcher which implements
    RMIRemoteSessionController. The constructor takes an OracleAS TopLink
    session as a parameter */
    controller = new RMIRemoteSessionControllerDispatcher (localTopLinkSession);
}
catch (RemoteException exception) {
    System.out.println("Error in invocation " + exception.toString());
}
return controller;
```

Sessions and the Cache

OracleAS TopLink automatically caches any data that is returned when a client reads an object. The cache resides with the session, which enables any associated client sessions to share the cache. This cache plays an important role in the performance of your application.

To define how the cache manages objects, specify a strategy for cache management in the OracleAS TopLink Mapping Workbench.

For more information, see "Working with Identity Maps" in the *Oracle Application Server TopLink Mapping Workbench User's Guide*.

Session Utilities

The OracleAS TopLink session provides several utilities to test and troubleshoot your application. This section introduces these tools and describes techniques for using them:

- Logging SQL and Messages
- Using the Profiler
- Using the Integrity Checker
- Using Exception Handlers

Logging SQL and Messages

OracleAS TopLink accesses the database using SQL strings that it generates internally. This feature enables applications to use the session methods or query objects without having to perform their own SQL translation.

If, for debugging purposes, you want to review a record of the SQL that is sent to the database, sessions provide these methods to log generated SQL to a writer. OracleAS TopLink disables SQL and message logging by default. To enable it, use the `logMessages()` method on the session. The default writer is a stream writer to `System.out`, but you can configure the log destination using the `setLog()` method on the session.

The session logs:

- Debug print statements
- Exceptions/error messages sent to system out

- Any other output sent to the system log

Logging Chained Exceptions

The logging chained exception facility enables you to log causality when one exception causes another as part of the standard stack back-trace. If you build your applications with JDK 1.4, causal chains appear automatically in your logs.

Logging and the Oracle Enterprise Manager

You can view OracleAS TopLink logs with all the other Oracle Application Server 10*g* log files using the Oracle Enterprise Manager.

For more information, see "Managing Diagnostic Log Files" in the *Oracle Application Server 10g Administrator's Guide*.

If you install OracleAS TopLink in the same Oracle Home directory as Oracle Application Server, OracleAS TopLink logs appear automatically with the other Oracle Application Server component log files in the Oracle Enterprise Manager. If you install OracleAS TopLink in a different Oracle Home directory, use the following procedure:

1. Locate the `toplink.xml` file in the *<ORACLE_HOME>*`\toplink\config\` directory.
2. Ensure that the `log path` tag reflects the location of your OracleAS TopLink log file, and is properly configured.

   For example:

   ```
   - <log path="toplink/config/toplink.log"
      componentId="TOPLINK" encoding="utf-8">
   ```

3. Copy the `toplink.xml` file to the following directory:

   <ORACLE_HOME>`\diagnostics\config\registration\`

Using the Profiler

The OracleAS TopLink Profiler is a high-level logging service. Instead of logging SQL statements, the Profiler logs a summary of each query you execute. The summary includes a performance breakdown of the query that enables you to identify performance bottlenecks. The Profiler also provides a report summarizing the query performance for an entire session.

Access Profiler reports and profiles through the **Profile** tab in the OracleAS TopLink Web Client, or create your own application or applet to view the Profiler logs.

For more information about the Web Client, see "OracleAS TopLink -- Web Client".

Using the Integrity Checker

When you connect a session or add descriptors to a session after connection, OracleAS TopLink initializes and validates the descriptor information. The integrity checker allows you to customize the validation process. The integrity checker offers the following configuration options:

Catch All Exceptions

This option specifies whether or not the integrity checker catches all exceptions in the session. The settings for this option are `catchExceptions` (the default setting), and `dontcatchExceptions`.

Catch Instantiation Policy Exceptions

This option catches only errors that are associated with instantiation policy, and:

- Throws the first error that it encounters, including the error's stack trace.
- Validates the state of the database schema to ensure that it matches the information in the descriptors.
- Disables the instance creation check.

Example 4-44 Using the Integrity Checker

```
session.getIntegrityChecker().checkDatabase();
session.getIntegrityChecker().catchExceptions();
session.getIntegrityChecker().dontCheckInstantiationPolicy();
session.login();
```

Using Exception Handlers

Exception handlers process database exceptions, generally to process connection timeouts or database failures. To use exception handlers, register an implementor of the `ExceptionHandler` interface with the session. If a database exception occurs during the execution of a query, the exception passes to the exception handler. The exception handler then either handles the exception, retries the query, or throws an unchecked exception.

For more information about exceptions, see Appendix C, "Error Codes and Messages".

Example 4-45 Implementing an Exception Handler

```
session.setExceptionHandler(newExceptionHandler(){
    public Object handleException(RuntimeException exception) {
        if ((exception instanceof DatabaseException) &&
            (exception.getMessage().equals("connection reset by peer."))) {
            DatabaseException dbex = (DatabaseException) exception;
            dbex.getAccessor().reestablishConnection
            (dbex.getSession());
            return dbex.getSession().executeQuery(dbex.getQuery());
        }
        throw exception;
    }
});
```

Note:

Unhandled exceptions must be re-thrown by the handler code.

Customizing Session Events

Sessions, such as database sessions, Unit of Work, client sessions, server sessions, and remote sessions raise *session events* for most session operations. Session events help you debug or coordinate the actions of multiple sessions.

This section illustrates how you customize session events, and discusses:

- Session Event Listeners
- Session Event Manager
- Implementing Events Using Java

Session Event Listeners

One approach to customizing session events is to create session event listeners that detect and respond to session events. To register objects as listeners for session events, implement the `SessionEventListener` interface and register it with the `SessionEventManager` using `addListener()`.

Table 4-14 Session Event Manager Events

Event	Description
`PreExecuteQuery`	Raised before the execution of every query on the session
`PostExecuteQuery`	Raised after the execution of every query on the session
`PreBeginTransaction`	Raised before a database transaction starts
`PostBeginTransaction`	Raised after a database transaction starts
`PreCommitTransaction`	Raised before a database transaction commits
`PostCommitTransaction`	Raised after a database transaction commits
`PreRollbackTransaction`	Raised before a database transaction rolls back
`PostRollbackTransaction`	Raised after a database transaction rolls back
`PreLogin`	Raised before the Session initializes and acquires connections
`PostLogin`	Raised after the Session initializes and acquires connections

Table 4-15 Unit of Work Events

Event	Description
`PostAcquireUnitOfWork`	Raised after a `UnitOfWork` is acquired

Event	Description
`PreCommitUnitOfWork`	Raised before a `UnitOfWork` commits
`PrepareUnitOfWork`	Raised after the a `UnitOfWork` flushes its SQL, but before it commits its transaction
`PostCommitUnitOfWork`	Raised after a `UnitOfWork` commits
`PreReleaseUnitOfWork`	Raised on a `UnitOfWork` before it releases
`PostReleaseUnitOfWork`	Raised on a `UnitOfWork` after it releases
`PostResumeUnitOfWork`	Raised on a `UnitOfWork` after it resumes

Table 4-16 Server Session and Client Session Events (Three-Tier Applications)

Event	Description
`PostAcquireClientSession`	Raised after a `ClientSession` is acquired
`PreReleaseClientSession`	Raised before releasing a `ClientSession`
`PostReleaseClientSession`	Raised after releasing a `ClientSession`
`PostConnect`	Raised after connecting to the database
`PostAcquireConnection`	Raised after acquiring a connection
`PreReleaseConnection`	Raised before releasing a connection

Table 4-17 Database Access Events

Event	Description
`OutputParametersDetected`	Raised after a stored procedure call with output parameters executes. This event enables you to retrieve a result set and output parameters from a single stored procedure.

Event	Description
MoreRowsDetected	Raised when a ReadObjectQuery detects more than one row returned from the database. This event can indicate a possible error condition in your application.

Session Event Manager

The session event manager handles information about session events. Applications register listeners with the session event manager to receive session event data.

Example 4-46 Registering a Listener

```
public void addSessionEventListener(SessionEventListener listener)
{
    // Register specified listener to receive events from mySession
    mySession.getEventManager().addListener(listener);
}
```

Example 4-47 Using the Session Event Adapter to Listen for Specific Session Events

```
...
    SessionEventAdapter myAdapter = new SessionEventAdapter() {
        // Listen for PostCommitUnitOfWork events
        public void postCommitUnitOfWork(SessionEvent event) {
        // Call my handler routine
    unitOfWorkCommitted();

    }
};
mySession.getEventManager().addListener(myAdapter);
...
```

Implementing Events Using Java

You can implement custom events and event handlers in Java code. The code in Example 4-48 checks for lock conflicts when the application builds an instance of `Employee` from information in the database.

Example 4-48 Implementing an Event in Code

```
/*In the employee class, declare the event method which will be invoked when the
event occurs */
public void postBuild(DescriptorEvent event) {
    // Uses object row to integrate with some application level locking service.
    if ((event.getRow().get("LOCKED")).equals("T")) {
        LockManager.checkLockConflict(this);
   }
}
```

OracleAS TopLink Support for Java Data Objects (JDO)

Java Data Objects (JDO) is an API for transparent database access. The JDO architecture is a standard API for data, both in local storage systems and enterprise information systems. It unifies access to heterogeneous systems, such as mainframe transaction processing, and database systems. JDO enables programmers to create Java code that accesses the underlying data store transparently and does not require database-specific code.

OracleAS TopLink provides basic JDO support based on the JDO specification. OracleAS TopLink support includes much of the JDO API, but does not require you to enhance or modify the class to leverage JDO.

This section includes information on:

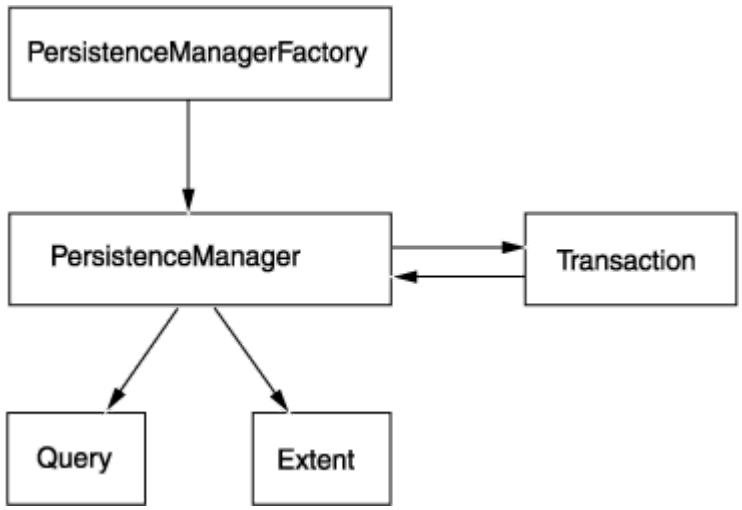
- Understanding the JDO API
- JDO Implementation
- Running the OracleAS TopLink JDO Example

Understanding the JDO API

The JDO API includes four main interfaces:

- The *PersistenceManagerFactory* is a factory that generates `PersistenceManagers`. It has a configuration and login API.
- The *PersistenceManager* is the main point of contact from the application. It provides an API for accessing the transaction, queries, and object life cycle API (`makePersistent`, `makeTransactional`, `deletePersistent`).
- The *Transaction* defines a basic begin, commit, rollback API.
- The *Query* defines the API to configure the query (filter, ordering, parameters, and variables) and to execute the query.

Figure 4-12 Understanding the JDO API



Text description of the illustration jdo_api.gif

JDO Implementation

OracleAS TopLink implements the `PersistenceManagerFactory`, `PersistenceManager`, and `Transaction` interfaces, and extends the query functionality to include the complete OracleAS TopLink query framework.

JDOPersistenceManagerFactory

To create a `JDOPersistenceManagerFactory`, call the constructor and include a session name string, or an OracleAS TopLink session or project. If you construct the factory from a project, then OracleAS TopLink creates a new database session and attaches it to the `PersistenceManager` every time you obtain the `PersistenceManager` with the `getPersistenceManager` method.

The `PersistenceManager` is not multi-threaded. In a multi-threaded application, assign each thread its own `PersistenceManager`. In addition, construct the `JDOPersistenceManagerFactory` from a server session, rather than a database session or project. Doing this enables you to use the lightweight client session and more scalable connection pooling.

Creating a JDOPersistenceManagerFactory

Example 4-49 illustrates how to create a factory from an OracleAS TopLink session named `jdoSession`. A session manager manages a singleton instance of the OracleAS TopLink server session or database session.

For more information, see "Session Manager".

Example 4-49 Creating a JDOPersistenceManagerFactory

```
JDOPersistenceManagerFactory factory= new
JDOPersistenceManagerFactory("jdoSession");
/*Create a persistence manager factory from an instance of OracleAS TopLink
ServerSession or DatabaseSession that is managed by the user */
ServerSession session = (ServerSession) project.createServerSession();
JDOPersistenceManagerFactory factory= new JDOPersistenceManagerFactory(session);
/* Create a persistence manager factory with ties to a DatabaseSession that is
created from OracleAS TopLink project */
JDOPersistenceManagerFactory factory= new JDOPersistenceManagerFactory(new
EmployeeProject());
```

Obtaining PersistenceManager

To create new PersistenceManagers, call the `getPersistentManager` method. If you construct the factory from a Project instance, use the `getPersistentManager(String userid, String password)` method to configure the `userid` and `password`.

Reference

Table 4-18 summarizes the most common public methods for `PersistenceManagerFactory`. For more information about the available methods for `PersistenceManagerFactory`, see the *Oracle Application Server TopLink API Reference*.

Table 4-18 Elements for Persistence Manager Factory

Method Name	Description
`JDOPersistenceManagerFactory() persistence`	Constructs a factory from a session manager session
`JDOPersistenceManagerFactory(String sessionName)`	Constructs a factory from the named session
`JDOPersistenceManagerFactory(Session session)`	Constructs a factory from a user session

Method Name	Description
`JDOPersistenceManagerFactory(Project project)`	Constructs a factory from a project
`getIgnoreCache()setIgnoreCache (boolean ignoreCache)`	Query mode that specifies whether cached instances are considered when evaluating the filter expression. The default is set to FALSE.
`getNontransactionalRead()setNontransactionalRead (boolean nontransactionalRead)`	Transaction mode that allows you to read instances outside a transaction. The default is set to FALSE.

`getConnectionUserName()`
`setConnectionUserName(String userName)`
`getConnectionPassword()`
`setConnectionPassword(String password)`
`getConnectionURL()`
`setConnectionURL(String URL)`
`getConnectionDriverName()`
`setConnectionDriverName(String driverName)`
	Available settings if the factory is constructed from an OracleAS TopLink project.
	
	Derives the default user name, password, URL, driver from project login.

`getPersistenceManager()` `getPersistenceManager(String userid, String password)`	Accesses `PersistenceManager`, and sets the user ID and password if the factory is constructed from an OracleAS TopLink project (uses default values in the absence of a project). Derives the default user ID, password from session login, or project login.
`getProperties()`	Nonconfigurable properties
`supportedOptions()`	Collection of supported option String

JDOPersistenceManager

The `JDOPersistenceManager` class is the factory for the `Query` interface and contains methods to access transactions, and manage the persistent life cycle instances.

Inserting JDO objects

To make new JDO objects persistent, use the `makePersistent()` or `makePersistentAll()` method. If you do not manually begin the transaction, then OracleAS TopLink begins and commits the transaction when you invoke either `makePersistent()` or `makePersistentAll()`. If the object is already persisted, then calling these methods has no effect.

Example 4-50 Persist a New Employee Named Bob Smith

```
Server serverSession = new EmployeeProject().createServerSession();
PersistenceManagerFactory factory = new
JDOPersistenceManagerFactory(serverSession);
PersistenceManager manager = factory.getPersistenceManager();
Employee employee = new Employee();
employee.setFirstName("Bob");
employee.setLastName("Smith");
manager.makePersistent(employee);
```

Updating JDO Objects

To modify JDO objects within a transaction context, begin and commit a transactional object manually. A transactional object is an object that is subject to the transaction boundary. Use one of the following methods to obtain transactional objects:

- Using `getObjectById()`
- Executing a transactional-read query
- Using the OracleAS TopLink extended API `getTransactionalObject()`

OracleAS TopLink executes the transactional-read query when the `nontransactionalRead` flag of the current transaction is false. To obtain the current transaction from the `PersistenceManager`, call `currentTransaction()`.

Example 4-51 Update an Employee

This example illustrates how to add a new phone number to an employee object, modify the address, and increase the salary by 10 percent.

```
Transaction transaction = manager.currentTransaction();
if(!transaction.isActive()) {
transaction.begin();
}
// Get the transactional instance of the employee
Object id = manager.getTransactionalObjectId(employee);
Employee transactionalEmployee = manager.getObjectById(id, false);
transactionalEmployee.getAddress().setCity("Ottawa");
transactionalEmployee.setSalary((int) (employee.getSalary() * 1.1));
transactionalEmployee.addPhoneNumber(new PhoneNumber("fax", "613", "3213452"));

transaction.commit();
```

Deleting Persistent Objects

To delete JDO objects, use either `deletePersistent()` or `deletePersistentAll()`. The objects need not be transactional. If you do not manually begin the transaction, then OracleAS TopLink begins and commits the transaction when you invoke either `deletePersistent ()` or `deletePersistentAll ()`.

Deleting objects using `deletePersistent()` or `deletePersistentAll()` is similar to deleting objects using a Unit of Work. When you delete an object, you also automatically delete its privately owned

parts, because they cannot exist without their owner. At commit time, OracleAS TopLink generates SQL to delete the objects, taking database constraints into account.

When you delete an object, set references to the deleted object to null or remove them from the collection, and modify references to the object using its transactional instance. This ensures that the object model reflects the change.

Example 4-52 Deleting a Team Leader from a Project

```
Transaction transaction = manager.currentTransaction();
if(!transaction.isActive()) {
transaction.begin();
}
Object id = manager.getTransactionalObjectId(projectNumber);
Project transactionalProject = (Project) manager.getObjectById(id);
Employee transactionalEmployee = transactionalProject.getTeamLeader();
// Remove team leader from the project
transactionalProject.setTeamLeader(null);
// Remove owner that is the team leader from phone numbers
for(Enumeration enum = transactionalEmployee.getPhoneNumbers().elements();
enum.hasMoreElements();) {
     ((PhoneNumber) enum.nextElement()).setOwner(null);
}
manager.deletePersistent(transactionalEmployee);
transaction.commit();
```

Example 4-53 Deleting a Phone Number

```
Transaction transaction = manager.currentTransaction();
if(!transaction.isActive()) {
transaction.begin();
}
Object id = manager.getTransactionalObjectId(phoneNumber);
PhoneNumber transactionalPhoneNumber = (PhoneNumber) manager.getObjectById(id);
transactionalPhoneNumber.getOwner().getPhoneNumbers().remove(transactionalPhoneN
umber);
manager.deletePersistent(phoneNumber);
transaction.commit();
```

Obtaining Query

OracleAS TopLink does not support the JDO Query language, but includes support within JDO for the more advanced OracleAS TopLink query framework.

For more information about the OracleAS TopLink query framework, see Chapter 6, "Queries".

A key difference is that the JDO query language requires returned results to be a collection of candidate JDO instances (either a `java.util.Collection`, or an Extent). Conversely, the return type in OracleAS TopLink depends on the type of query. For example, if you use a `ReadAllQuery`, the result is a `Vector`.

The following APIs support for the query factory:

- Standard API:

  ```
  newQuery();
  newQuery(Class persistentClass);
  ```

- OracleAS TopLink extended API:

  ```
  newQuery(Class persistentClass, Expression expressionFilter);
  ```

Note:

If you obtain Query from a different `newQuery()` API, this can result in a `JDOUserException`, or the creation of the query from the supported API.

You create a `ReadAllQuery` with the query instance by default.

Reference

Table 4-19 and Table 4-20 summarize the most common public methods for the Query API and OracleAS TopLink extended API. For more information about the available methods for the Query API and OracleAS TopLink extended API, see the *Oracle Application Server TopLink API Reference*.

Table 4-19 Elements for Query API

Method Name	Description
`close()`	Releases resource to allow garbage collection
`currentTransaction()`	Specifies current transaction
`deletePersistent (Object object)deletePersistentAll (Collection objects)deletePersistentAll (java.lang.Object[] objects)`	Deletes objects
`evict(Object object)evictAll()evictAll (Collection objects)evictAll(Object[] objects)`	Marks objects as no longer needed in the cache
`getExtent(Class queryClass, boolean readSubclasses)`	Specifies extent
`getIgnoreCache()setIgnoreCache(boolean ignoreCache)`	Sets cache mode for queries. The default is set to ignore cache from the persistence manager factory.

Method Name	Description

`getObjectById`
 	
`(Object object, boolean validate)`
 	
`getTransactionalObjectId(Object object)`
	Obtains transactional state of object

`isClosed()`	Closes the `PersistenceManager` instance

`makePersistent(Object object)`
 	
`makePersistentAll`
 	
`(Collection objects)makePersistentAll`
 	
`(Object[] objects)`
	Inserts persistent objects

`makeTransactional`
 	
`(Object object)makeTransactionalAll`
 	
`(Collection objects)makeTransactionalAll`
 	
`(Object[] objects)`
	Registers objects to Unit of Work, making them subject to transactional boundaries

`newQuery()newQuery(Class queryClass)`	Creates new query factory

`refresh(Object object)refreshAll()refreshAll`
 	
`(Collection objects)refreshAll(Object[] objects)`
	Refreshes objects

Table 4-20 Elements for OracleAS TopLink Extended API

Method Name	Description
`getTransactionalObject(Object object)`	Obtains transactional object
`newQuery(Class queryClass, Expression expression)`	Creates query factory

Method Name	Description
`readAllObjects(Class domainClass)readAllObjects`	Reads objects
`(Class domainClass)readObject`	
`(Class domainClass, Expression expression)`	

JDOQuery

The `JDOQuery` class implements the `JDOQuery` interface. It defines the API to configure the query (filter, ordering, parameters, and variables) and to execute the query. OracleAS TopLink extends the query functionality to include the full OracleAS TopLink query framework.

For more information about the OracleAS TopLink query framework, see Chapter 6, "Queries".

Users can customize the query to use advanced features, such as batch reading, stored procedure calls, partial object reading, and query by example. OracleAS TopLink does not support the JDO query language, but users can employ either SQL or EJB QL in the `JDOQuery` interface.

Each `JDOQuery` instance is associated with an OracleAS TopLink query. To obtain a `JDOQuery` from the `PersistenceManager`, call a supported `newQuery` method. OracleAS TopLink creates a new `ReadAllQuery` and associates it with the query. Call `asReadObjectQuery()`, `asReadAllQuery()`, or `asReportQuery` to set the JDO Query OracleAS TopLink query to a specific type.

Customizing the Query Using the OracleAS TopLink Query Framework

The OracleAS TopLink query framework provides most of its functionality as a public API. To create a customized OracleAS TopLink query and associate it with the JDO Query, call the `setQuery()` method to build complex functionality into your queries.

Customized OracleAS TopLink queries give you the complete functionality of the OracleAS TopLink query framework. For example, use a `DirectReadQuery` with custom SQL to read the ID column of the employee.

Note:

OracleAS TopLink extended APIs support a specific OracleAS TopLink query type. To avoid exceptions, match the API to the correct query type. See Table 4-21 for correct usage.

Example 4-54 Use a ReadAllQuery to Read All Employees Who Live in New York

```
Expression expression = new
ExpressionBuilder().get("address").get("city").equal("New York");
Query query = manager.newQuery(Employee.class, expression);
Vector employees = (Vector) query.execute();
```

Example 4-55 Use a ReadObjectQuery to Read the Employee Named Bob Smith

```
Expression exp1 = new ExpressionBuilder().get("firstName").equal("Bob");
Expression exp2 = new ExpressionBuilder().get("lastName").equal("Smith ");
JDOQuery jdoQuery = (JDOQuery) manager.newQuery(Employee.class);
jdoQuery.asReadObjectQuery();
jdoQuery.setFilter(exp1.and(exp2));
Employee employee = (Employee) jdoQuery.execute();
```

Example 4-56 Use a ReportQuery to Report Employee's Salary

```
JDOQuery jdoQuery = (JDOQuery) manager.newQuery(Employee.class);
jdoQuery.asReportQuery();
jdoQuery.addCount();
jdoQuery.addMinimum("min_salary",jdoQuery.getExpressionBuilder().get("salary"));
jdoQuery.addMaximum("max_salary",jdoQuery.getExpressionBuilder().get("salary"));
jdoQuery.addAverage("average_salary",jdoQuery.getExpressionBuilder().get("salary"));
// Return a vector of one DatabaseRow that contains reported info
Vector reportQueryResults = (Vector) jdoQuery.execute();
```

Example 4-57 Use a Customized DirectReadQuery to Read Employee 's id column

```
DirectReadQuery TopLinkQuery = new DirectReadQuery();
topLinkQuery.setSQLString("SELECT EMP_ID FROM EMPLOYEE");
JDOQuery jdoQuery = (JDOQuery) manager.newQuery();
jdoQuery.setQuery(topLinkQuery);
// Return a Vector of DatabaseRows that contain ids
Vector ids = (Vector)jdoQuery.execute(query);
```

Reference

Table 4-21 and Table 4-22 summarize the most common public methods for the JDO Query API and OracleAS TopLink extended API. For more information about the available methods for the JDO Query API and OracleAS TopLink extended API, see the *Oracle Application Server TopLink API Reference*.

Table 4-21 Elements for JDO Query API

Method Name	Description
`close(Object queryResult)`	Closes cursor result
`declareParameters(String parameters)`	Declares query parameters
`execute()execute(Object arg1)execute (Object arg1, Object arg2)execute (Object arg1, Object arg2, Object arg3)executeWithArray (java.lang.Object[] arg1)executeWithMap(Map arg1)`	Executes query
`getIgnoreCache()setIgnoreCache(boolean ignoreCache)`	Sets cache mode for query result
`getPersistenceManager()`	`PersistenceManager`

Method Name	Description
setClass(Class queryClass)	ReadObjectQuery, ReadAllQuery, ReportQuery
setOrdering(String ordering)	ReadAllQuery

Table 4-22 Elements for OracleAS TopLink Extended JDO API

Method Name	Description
asReadAllQuery()asReadObjectQuery() asReportQuery()	Converts the query
getQuery()setQuery(DatabaseQuery newQuery)	Accesses the OracleAS TopLink query. The default is set to ReadAllQuery.
acquireLocks() acquireLocksWithoutWaiting() addJoinedAttribute(String attributeName) addJoinedAttribute(Expression attributeExpression) addPartialAttribute(String attributeName) addPartialAttribute(Expression attributeExpression) checkCacheOnly() dontAcquireLocks() dontRefreshIdentityMapResult() dontRefreshRemoteIdentityMapResult() getExampleObject() getExpressionBuilder() setQueryByExampleFilter(Object exampleObject) setQueryByExamplePolicy(QueryByExamplePolicy newPolicy) setShouldRefreshIdentityMapResult(boolean shouldRefreshIdentityMapResult) shouldRefreshIdentityMapResult()	ReadObjectQuery, ReadAllQuery, ReportQuery

Method Name	Description
checkCacheByExactPrimaryKey()	ReadObjectQuery
checkCacheByPrimaryKey()	
checkCacheThenDatabase()	
conformResultsInUnitOfWork()	
getReadObjectQuery()	
addAscendingOrdering(String queryKeyName)	ReadAllQuery
addDescendingOrdering(String queryKeyName)	
addOrdering(Expression orderingExpression)	
addBatchReadAttribute(String attributeName)	
addBatchReadAttribute(Expression attributeExpression)	
addStandardDeviation(String itemName)	
addStandardDeviation(String itemName,Expression attributeExpression)	
addSum(String itemName)	
addSum(String itemName, Expression attributeExpression)	
addVariance(String itemName)	
addVariance(String itemName, Expression attributeExpression)	
getReadAllQuery()	
useCollectionClass(Class concreteClass)	
useCursoredStream()	
useCursoredStream(int initialReadSize, int pageSize)	
useCursoredStream(int initialReadSize, int pageSize, ValueReadQuery sizeQuery)	
useDistinct()useMapClass(Class concreteClass, String methodName)	
useScrollableCursor()	
useScrollableCursor(int pageSize)	

Method Name	Description
addAttribute(String itemName)	Query arguments
addAttribute(String itemName, Expression attributeExpression)	
addAverage(String itemName)	
addAverage(String itemName, Expression attributeExpression)	
ddCount()	
addCount(String itemName)	
addCount(String itemName, Expression attributeExpression)	
addGrouping(String attributeName)	
addGrouping(Expression expression)	
addItem(String itemName, Expression attributeExpression)	
addMaximum(String itemName)	
addMaximum(String itemName, Expression attributeExpression)	
addMinimum(String itemName)	
addMinimum(String itemName, Expression attributeExpression)	
getReportQuery()	
addArgument(String argumentName)	DatabaseQuery
bindAllParameters()	
cacheStatement()	
cascadeAllParts()	
cascadePrivateParts()	
dontBindAllParameters()	
dontCacheStatement()	

Method Name	Description
dontCascadeParts()	
dontCheckCache()	
dontMaintainCache()	
dontUseDistinct()	
getQueryTimeout()getReferenceClass()	
getSelectionCriteria()	
refreshIdentityMapResult()	
setCall(Call call)	
setEJBQLString(String ejbqlString)	
setFilter(Expression selectionCriteria)	
setQueryTimeout(int queryTimeout)	
setSQLString(String sqlString)	
setShouldBindAllParameters(boolean shouldBindAllParameters)	
setShouldCacheStatement(boolean shouldCacheStatement)	
setShouldMaintainCache(boolean shouldMaintainCache)	
shouldBindAllParameters()	
shouldCacheStatement()	
shouldCascadeAllParts()	
shouldCascadeParts()	
shouldCascadePrivateParts()	
shouldMaintainCache()	

JDOTransaction

The `JDOTransaction` class implements the `JDOTransaction` interface. It defines the basic begin, commit, and rollback APIs, and synchronization callbacks within the Unit of Work. It supports the optional nontransactional read JDO feature.

Read Modes

Set the read mode of a JDO transaction by calling the `setNontransactionalRead()` method.

Note:

To avoid exceptions, do not change the read mode while the transaction is active.

Here are the available read modes:

- *Nontransactional Read:* Nontransactional reads provide data from the database, but do not attempt to update the database with changes at commit time. This transaction mode is the `PersistenceManagerFactory` default. Nontransactional reads support nested Units of Work.

  When you execute queries in nontransactional read mode, their results are not subject to the transactional boundary. To update objects from the query results, modify objects in their transactional instances.

  To enable nontransactional read mode, set `setNontransactionalRead()` to true.

- *Transactional Read:* Transactional reads provide data from the database and write any changes to the database at commit time. When you use transactional read, OracleAS TopLink uses the same Unit of Work for all data store interaction (begin, commit, rollback). Because this can cause the cache to grow large over time, use this mode only with short-lived PersistenceManager instances. Doing this allows garbage collection on the Unit of Work.

  When you execute queries in transactional read mode, the results are transactional instances, subject to the transactional boundary. You can update objects from the result of a query that is executed in transactional mode.

  Because you use the same Unit of Work in this mode, the transaction is always active. You must release it when you change the read mode from transactional read to nontransactional read.

  Caution:

  Before you call the OracleAS TopLink extended API `release()` method, commit all changes to avoid losing the transaction.

To enable transactional read mode, set the `setNontransactionalRead()` flag to false.

Synchronization

You can register a Synchronization listener with the transaction. The transaction notifies the listener when the transaction is complete. Doing this returns the `beforeCompletion` and `afterCompletion` methods when the precommit and post-commit events of the Unit of Work trigger.

Running the OracleAS TopLink JDO Example

OracleAS TopLink includes an example application that illustrates some of the JDO functionality. You can locate the example in *<ORACLE_HOME>*\toplink\examples\foundation\jdo.