November 2013

Documentation for developers and administrators that describes how to integrate Oracle Coherence with TopLink Grid, EclipseLink JPA, Hibernate, Spring, Coherence*Web, GoldenGate HotCache, and WebLogic Portal.
# Contents

**Preface** ......................................................................................................................................................... xi  
  Audience ......................................................................................................................................................... xi  
  Documentation Accessibility ............................................................................................................................. xi  
  Related Documents ......................................................................................................................................... xi  
  Conventions .................................................................................................................................................... xii  

**What's New in This Guide** ............................................................................................................................ xiii  
  New and Changed Features for 12c (12.1.2) ....................................................................................................... xiii  
  Other Significant Changes in this Document for 12c (12.1.2) ........................................................................ xiii  

1 **Integrating TopLink Grid with Oracle Coherence**  
   What is TopLink Grid? ....................................................................................................................................... 1-1  
   What are the JPA on the Grid Configurations? ................................................................................................. 1-2  
   What are the Benefits of Using TopLink Grid with Oracle Coherence? .......................................................... 1-2  
   Required Files ................................................................................................................................................ 1-3  
   JPA on the Grid Configurations ......................................................................................................................... 1-3  
     Understanding JPA on the Grid ....................................................................................................................... 1-3  
     JPA on the Grid API ..................................................................................................................................... 1-4  
   Grid Cache Configuration ................................................................................................................................. 1-5  
     Reading Objects in Grid Cache Configuration .............................................................................................. 1-5  
     Writing Objects in Grid Cache Configuration .............................................................................................. 1-6  
   Grid Cache Configuration Examples .............................................................................................................. 1-7  
     Configuring the Cache for the Grid Cache Configuration ........................................................................... 1-7  
     Configuring an Entity for the Grid Cache Configuration .............................................................................. 1-8  
     Inserting Objects for the Grid Cache Configuration .................................................................................... 1-8  
     Querying Objects for the Grid Cache Configuration .................................................................................... 1-8  
   Grid Read Configuration .................................................................................................................................... 1-9  
     Reading Objects in Grid Read Configuration .............................................................................................. 1-9  
     Writing Objects in Grid Read Configuration .............................................................................................. 1-10  
   Grid Read Configuration Examples ............................................................................................................... 1-11  
     Configuring the Cache in Grid Read Configuration .................................................................................... 1-11  
     Reading Objects for the Grid Read Configuration ..................................................................................... 1-12  
     Inserting Objects for the Grid Read Configuration .................................................................................... 1-13  
     Querying Objects for the Grid Read Configuration .................................................................................... 1-13
Grid Entity Configuration................................................................................................. 1-13
Reading Objects in Grid Entity Configuration ............................................................... 1-14
Writing Objects in Grid Entity Configuration .................................................................. 1-14
Limitations on Writing Objects in Grid Entity Configuration ........................................... 1-14
Grid Entity Configuration Examples .............................................................................. 1-15
Configuring the Cache for the Grid Entity Configuration .................................................. 1-15
Configuring an Entity for the Grid Entity Configuration .................................................. 1-16
Persisting Objects for the Grid Entity Configuration ....................................................... 1-16
Querying Objects for the Grid Entity Configuration ......................................................... 1-16
Handling Grid Read and Grid Entity Failovers ................................................................. 1-17
Wrapping and Unwrapping Entity Relationships ............................................................. 1-17
Working with Queries ...................................................................................................... 1-17
Querying Objects by ID.................................................................................................... 1-18
Querying Objects with Criteria ......................................................................................... 1-18
Using Indexes in Queries ................................................................................................. 1-18
Limitations on Queries .................................................................................................... 1-19
EclipseLink Native ORM Configurations ......................................................................... 1-19
Understanding EclipseLink Native ORM ........................................................................ 1-19
API for EclipseLink Native ORM ...................................................................................... 1-20
Configuring an Amendment Method .............................................................................. 1-20
Configuring the Amendment Method in JDeveloper ....................................................... 1-21
Configuring the EclipseLink Native ORM Cache Store and Cache Loader ....................... 1-24
Using POF Serialization with TopLink Grid and Coherence ............................................ 1-25
Implement a Serialization Routine .................................................................................... 1-26
Define a Cache Configuration File .................................................................................... 1-28
Define a POF Configuration File ..................................................................................... 1-29
Best Practices .................................................................................................................. 1-32
Changing Compiled Java Classes with Byte Code Weaving ............................................ 1-32
Deferring Database Queries with Lazy Loading ............................................................. 1-33
Defining Near Caches for Applications Using TopLink Grid .......................................... 1-33
Ensuring Prefixed Cache Names Use Wildcard in Cache Configuration .......................... 1-34
Overriding the Default Cache Name ............................................................................... 1-35

2 Integrating JPA Using the Coherence API

Using TopLink Grid with Coherence Client Applications ................................................. 2-1
API for Coherence with TopLink Grid Configurations ...................................................... 2-2
Sample Cache Configuration File for Coherence with TopLink Grid ............................... 2-2
Sample Project for Using Coherence with TopLink Grid ................................................. 2-4
Using Third Party JPA Providers ...................................................................................... 2-4
API for Native Coherence JPA CacheStore and CacheLoader ......................................... 2-4
Steps to Use a Third Party JPA Provider and Native Coherence JPA API .......................... 2-5
Obtain a JPA Provider Implementation ............................................................................ 2-5
Configure a Coherence JPA Cache Store ......................................................................... 2-5
Map the Persistent Classes ............................................................................................. 2-5
Configure JPA .................................................................................................................. 2-5
Configure a Coherence Cache for JPA ............................................................................. 2-6
Configure the Persistence Unit ....................................................................................... 2-7
3 Integrating Coherence Applications with Coherence*Web
Merging Coherence Cache and Session Information ......................................................... 3-1

4 Integrating Hibernate and Coherence
API for HibernateCacheStore and HibernateCacheLoader ........................................... 4-1
Using Hibernate as a Cache Store for Coherence .............................................................. 4-2
  Configuration Requirements .............................................................................................. 4-2
  Configuring a Hibernate Cache Store Constructor ......................................................... 4-2
  Creating a Hibernate Cache Store .................................................................................... 4-5
    Reentrant Calls ................................................................................................................. 4-5
  Extending the Hibernate Cache Store Functionality ......................................................... 4-5
  JDBC Isolation Level ........................................................................................................ 4-6
  Fault-Tolerance for Hibernate Cache Store Operations ................................................... 4-6
  Using Fully Cached Data Sets .......................................................................................... 4-6
    Distributed Queries ......................................................................................................... 4-6
    Detached Processing ....................................................................................................... 4-6
Using Coherence as the Hibernate L2 Cache Provider ...................................................... 4-7
  Configuring Coherence as the Hibernate L2 Cache .......................................................... 4-7
  Specifying a Coherence Cache Topology ........................................................................ 4-8
  Choosing a Cache Concurrency Strategy ......................................................................... 4-8
  Query Cache ..................................................................................................................... 4-9
  Fault-Tolerance for Hibernate L2 Caches ....................................................................... 4-9
  Deployment ....................................................................................................................... 4-9

5 Integrating Spring with Coherence

6 Enabling ECID in Coherence Logs

7 Integrating with Oracle Coherence GoldenGate HotCache
Overview ............................................................................................................................. 7-1
How Does HotCache Work? ............................................................................................... 7-2
  How the GoldenGate Java Adapter uses JPA Mapping Metadata ...................................... 7-4
  Supported Database Operations ....................................................................................... 7-4
Prerequisites ....................................................................................................................... 7-4
Configuring GoldenGate .................................................................................................... 7-5
  Monitor Table Changes ...................................................................................................... 7-5
  Filter Changes Made by the Current User ......................................................................... 7-6
Configuring HotCache ....................................................................................................... 7-7
  Create a Properties File with GoldenGate for Java Properties ...................................... 7-7
  Add Java Boot Options to the Properties File .................................................................. 7-8
    Java Classpath Files ...................................................................................................... 7-9
    HotCache-related Properties ......................................................................................... 7-9
    Coherence-related Properties ...................................................................................... 7-9
  Logging Properties ......................................................................................................... 7-10
  Provide Coherence*Extend Connection Information ...................................................... 7-10
Configuring the GoldenGate Java Client ................................. 7-11
Edit the GoldenGate Java Client Extracts File ......................... 7-12
Using Portable Object Format with HotCache ......................... 7-13
Enabling Wrapper Classes for TopLink Grid Applications .......... 7-13

8 Integrating WebLogic Portal and Coherence

P13N CacheProvider SPI Implementation .................................. 8-1
Sharing Data Between WSRP-Federated Portals Using Coherence 8-2
List of Examples

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>Configuring the Cache in Grid Cache Configuration</td>
<td>1-7</td>
</tr>
<tr>
<td>1–2</td>
<td>Configuring the Entity in Grid Cache Configuration</td>
<td>1-8</td>
</tr>
<tr>
<td>1–3</td>
<td>Inserting Objects in Grid Cache Configuration</td>
<td>1-8</td>
</tr>
<tr>
<td>1–4</td>
<td>Querying Objects in Grid Cache Configuration</td>
<td>1-8</td>
</tr>
<tr>
<td>1–5</td>
<td>Configuring the Cache in Grid Read Configuration</td>
<td>1-11</td>
</tr>
<tr>
<td>1–6</td>
<td>Configuring the Entity in Grid Read Configuration</td>
<td>1-12</td>
</tr>
<tr>
<td>1–7</td>
<td>Inserting Objects in Grid Read Configuration</td>
<td>1-13</td>
</tr>
<tr>
<td>1–8</td>
<td>Querying Objects in Grid Read Configuration</td>
<td>1-13</td>
</tr>
<tr>
<td>1–9</td>
<td>Configuring the Cache in Grid Entity Configuration</td>
<td>1-15</td>
</tr>
<tr>
<td>1–10</td>
<td>Configuring an Entity in Grid Entity Configuration</td>
<td>1-16</td>
</tr>
<tr>
<td>1–11</td>
<td>Persisting Objects in Grid Entity Configuration</td>
<td>1-16</td>
</tr>
<tr>
<td>1–12</td>
<td>Querying Objects in Grid Entity Configuration</td>
<td>1-16</td>
</tr>
<tr>
<td>1–13</td>
<td>Unwrapping an Entity</td>
<td>1-17</td>
</tr>
<tr>
<td>1–14</td>
<td>Exposing a Coherence Query Index to TopLink Grid</td>
<td>1-18</td>
</tr>
<tr>
<td>1–15</td>
<td>Configuration for an Integrated EclipseLinkNativeCacheStore</td>
<td>1-24</td>
</tr>
<tr>
<td>1–16</td>
<td>Configuration for an Integrated EclipseLinkNativeCacheLoader</td>
<td>1-25</td>
</tr>
<tr>
<td>1–17</td>
<td>Sample Entity Class that Implements PortableObject</td>
<td>1-26</td>
</tr>
<tr>
<td>1–18</td>
<td>Sample Cache Configuration File</td>
<td>1-28</td>
</tr>
<tr>
<td>1–19</td>
<td>Simplified POF Configuration File</td>
<td>1-30</td>
</tr>
<tr>
<td>1–20</td>
<td>Sample POF Configuration File with Definitions for TopLink Classes</td>
<td>1-31</td>
</tr>
<tr>
<td>1–21</td>
<td>Session Customizer to Prepend</td>
<td>1-34</td>
</tr>
<tr>
<td>2–1</td>
<td>Configuring the Cache for Coherence with TopLink Grid</td>
<td>2-3</td>
</tr>
<tr>
<td>2–2</td>
<td>Sample persistence.xml File for JPA</td>
<td>2-6</td>
</tr>
<tr>
<td>2–3</td>
<td>Assigning Named Caches to a JPA Caching Scheme</td>
<td>2-6</td>
</tr>
<tr>
<td>4–1</td>
<td>Coherence Cache Configuration File for Hibernate</td>
<td>4-3</td>
</tr>
<tr>
<td>4–2</td>
<td>Coherence Cache Configuration File that Uses [cache-name] Macro</td>
<td>4-4</td>
</tr>
<tr>
<td>4–3</td>
<td>Sample coherence-cache-config.xml File with Generalized Mappings</td>
<td>4-4</td>
</tr>
<tr>
<td>4–4</td>
<td>Specifying a Coherence Provider Class</td>
<td>4-7</td>
</tr>
<tr>
<td>6–1</td>
<td>Using a DMS Context in Coherence Client Code</td>
<td>6-1</td>
</tr>
<tr>
<td>7–1</td>
<td>Mapping Instances of Employee Class to Rows with Java Code</td>
<td>7-2</td>
</tr>
<tr>
<td>7–2</td>
<td>Simplified Java Code for Mapping Instances of Employee Class to Rows</td>
<td>7-3</td>
</tr>
<tr>
<td>7–3</td>
<td>Mapping Instances of Employee Class to Rows with XML</td>
<td>7-3</td>
</tr>
<tr>
<td>7–4</td>
<td>Simplified XML for Mapping Instances of Employee Class to Rows</td>
<td>7-3</td>
</tr>
<tr>
<td>7–5</td>
<td>Sample Commands to Create a User, Grant Permissions, and Enable Logging</td>
<td>7-5</td>
</tr>
<tr>
<td>7–6</td>
<td>Sample GoldenGate Java Adapter ggsci Script to Monitor Table Changes</td>
<td>7-6</td>
</tr>
<tr>
<td>7–7</td>
<td>Sample Extract .prm File for the GoldenGate Java Adapter</td>
<td>7-6</td>
</tr>
<tr>
<td>7–8</td>
<td>.properties File for a HotCache Project</td>
<td>7-8</td>
</tr>
<tr>
<td>7–9</td>
<td>Coherence*Extend Section of a Client Cache Configuration File</td>
<td>7-10</td>
</tr>
<tr>
<td>7–10</td>
<td>Coherence*Extend Section of a Server Cache Configuration File</td>
<td>7-11</td>
</tr>
<tr>
<td>7–11</td>
<td>Sample .prm Parameter File for GoldenGate for Java Client</td>
<td>7-12</td>
</tr>
<tr>
<td>7–12</td>
<td>Sample POF Configuration File for HotCache</td>
<td>7-13</td>
</tr>
</tbody>
</table>
List of Figures

1–1  *JPA on the Grid* Approach ........................................................................................................ 1-4
1–2  Reading Objects in Grid Cache Configuration ........................................................................... 1-6
1–3  Writing and Persisting Objects in grid Cache Configuration .................................................. 1-7
1–4  Reading Objects with a Query ................................................................................................... 1-10
1–5  Writing and Persisting Objects in Grid Read Configuration ...................................................... 1-11
1–6  Writing and Persisting Objects in Grid Entity Configuration ................................................... 1-14
1–7  tlMap Descriptors in the JDeveloper Structure Pane .................................................................. 1-21
1–8  Advanced Properties Dialog Box .............................................................................................. 1-22
1–9  After Load Tab for a TopLink Descriptor ................................................................................... 1-23
1–10 Searching for the Class containing the Amendment Method ................................................ 1-23
1–11 Selecting the Amendment Method ............................................................................................ 1-24
2–1  Coherence with TopLink Grid Approach .................................................................................. 2-2
7–1  How HotCache Propagates Database Changes to the Cache .................................................. 7-2
7–2  EMPLOYEE Table Before and After an Update ......................................................................... 7-4
**List of Tables**

<table>
<thead>
<tr>
<th></th>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>TopLink Grid Classes to Build JPA on the Grid Applications</td>
<td>1-4</td>
<td></td>
</tr>
<tr>
<td>1–2</td>
<td>EclipseLink Classes for Native ORM Configurations</td>
<td>1-20</td>
<td></td>
</tr>
<tr>
<td>2–1</td>
<td>TopLink Grid Classes to build Coherence with TopLink Grid Applications</td>
<td>2-2</td>
<td></td>
</tr>
<tr>
<td>2–2</td>
<td>JPA-Related CacheStore and CacheLoader API Included with Coherence</td>
<td>2-4</td>
<td></td>
</tr>
<tr>
<td>4–1</td>
<td>Hibernate Cache Store and Cache Loader Classes</td>
<td>4-2</td>
<td></td>
</tr>
</tbody>
</table>
Oracle Coherence (Coherence) is a JCache-compliant in-memory caching and data management solution for clustered Java Platform Enterprise Edition (Java EE) applications and application servers. Coherence makes sharing and managing data in a cluster as simple as it is on a single server. It accomplishes this by coordinating updates to the data using clusterwide concurrency control, replicating and distributing data modifications across the cluster using the highest performing clustered protocol available, and delivering notifications of data modifications to any servers that request them. Developers can take advantage of Coherence features using the standard Java collections API to access and modify data, and use the standard JavaBeans event model to receive data change notifications.

**Audience**

This guide is for software developers and architects who will be integrating Coherence with TopLink-Grid, JPA, Hibernate, Spring, Coherence GoldenGate HotCache, and WebLogic Portal.

**Documentation Accessibility**

For information about Oracle’s commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

**Access to Oracle Support**

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

**Related Documents**

For more information about Oracle Coherence, see the following:

- Oracle Fusion Middleware Developing Applications with Oracle Coherence
- Oracle Fusion Middleware Developing Remote Clients for Oracle Coherence
- Oracle Fusion Middleware Tutorial for Oracle Coherence
- Oracle Fusion Middleware Administering HTTP Session Management with Oracle Coherence*Web
Oracle Fusion Middleware Managing Oracle Coherence
Oracle Fusion Middleware Administering Oracle Coherence
Oracle Fusion Middleware Securing Oracle Coherence
Oracle Fusion Middleware Java API Reference for Oracle Coherence
Oracle Fusion Middleware .NET API Reference for Oracle Coherence
Oracle Fusion Middleware C++ API Reference for Oracle Coherence

Conventions

The following text conventions are used in this guide:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><code>monospace</code></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
What's New in This Guide

The following topics introduce the new and changed features of Oracle Coherence and other significant changes that are described in this guide, and provides pointers to additional information. This document is the new edition of the formerly titled Oracle Coherence Integration Guide for Oracle TopLink with Coherence Grid.

New and Changed Features for 12c (12.1.2)

Oracle Coherence 12c (12.1.2) includes the following new and changed features for this document.

- Oracle Coherence GoldenGate HotCache (HotCache) allows changes to the database to be propagated to objects in the Coherence cache. See Chapter 7, "Integrating with Oracle Coherence GoldenGate HotCache."

Other Significant Changes in this Document for 12c (12.1.2)

For 12c (12.1.2), this guide has been updated in several ways. Following are the sections that have been added or changed.

- Oracle Coherence Integration Guide for Oracle TopLink with Coherence Grid has been removed from the 12c (12.1.2) release. The material in this documentation now appears in Chapter 1, "Integrating TopLink Grid with Oracle Coherence."

- The documentation for integrating Spring with Coherence has been moved to Coherence Community projects. Coherence Community projects provide example implementations for commonly used design patterns based on Oracle Coherence. See the following URL:

  http://java.net/projects/cohspr/
Integrating TopLink Grid with Oracle Coherence

This chapter describes how Oracle TopLink Grid enables you to scale out Java Persistence API (JPA) applications using Oracle Coherence. TopLink Grid provides applications with a number of options on how they can scale, ranging from using Coherence as a distributed shared (L2) cache up to directing JPQL queries to Coherence for parallel execution across the grid to reduce database load. With TopLink Grid, you do not have to rewrite your applications to scale out. You can use your investment in JPA, and still take advantage of the scalability of Coherence.

This chapter contains the following sections:

- What is TopLink Grid?
- Required Files
- JPA on the Grid Configurations
- EclipseLink Native ORM Configurations
- Using POF Serialization with TopLink Grid and Coherence
- Best Practices

1.1 What is TopLink Grid?

Oracle TopLink Grid is a feature of Oracle TopLink that provides integration between the EclipseLink JPA and Coherence. Standard JPA applications interact directly with their primary data store, typically a relational database. However, with TopLink Grid you can store some or all of your domain model in the Coherence data grid. This configuration is also known as JPA on the Grid.

You can easily configure TopLink Grid to use Coherence as the primary data store, execute queries against the grid, and allow Coherence to manage the persistence of new and modified data. Coherence provides the layer between JPA and the data store, where direct database calls can be offloaded from every application instance. This makes it possible for clustered application deployments to scale beyond the bounds of standard database operations.

The Oracle TopLink Grid page on the Oracle Technology Network provides additional information and code examples for Coherence for TopLink Grid.

1.1.1 What are the **JPA on the Grid Configurations**?

These are the typical **JPA on the Grid** configurations that applications can use:

- **Grid Cache configuration**, which uses Coherence as the TopLink L2 (shared) cache. This configuration applies the Coherence data grid to JPA applications that rely on database-hosted data that cannot be entirely preloaded into a Coherence cache. Some reasons why it might not be able to be preloaded include extremely complex queries that exceed the feature set of Coherence Filters, third-party database updates that create stale caches, reliance on native SQL queries, stored procedures or triggers, and so on.

  In this configuration, you can scale TopLink up into large clusters while avoiding the requirement to coordinate local L2 caches. Updates made to entities are available in all Coherence cluster members immediately, upon committing a transaction. For more information, see "Grid Cache Configuration" on page 1-5.

- **Grid Read configuration**, which is optimal for entities that require fast access to large amounts of (fairly stable) data and must write changes synchronously to the database. In these entities, cache warming could be used to populate the Coherence cache, but individual queries could also be directed to the database if necessary. For more information, see "Grid Read Configuration" on page 1-9.

- **Grid Entity configuration**, which is optimal for applications that require fast access to large amounts of (fairly stable) data and perform relatively few updates. This configuration can be combined with a Coherence cache store using write-behind to improve application response time by performing database updates asynchronously. For more information, see "Grid Entity Configuration" on page 1-13.

1.1.2 What are the Benefits of Using TopLink Grid with Oracle Coherence?

TopLink Grid provides the following benefits:

- Simple application configuration using annotations or XML configurations that align with standard JPA.

- The ability to store complex object graphs with relationships in Coherence.

- The ability to selectively choose which entities are stored in the grid and which are stored directly in the backing database.

- Allows you to execute JPQL queries in the Grid or directly against the database.

- Allows you to store entities with both eager and lazy relationships into Coherence.

TopLink Grid integrates the EclipseLink JPA implementation with Oracle Coherence and provides these development approaches:

- You can build applications using JPA and transparently use the power of the data grid for improved scalability and performance. In this **JPA on the Grid** approach, TopLink Grid provides a set of cache and query configuration options that allow you to control how EclipseLink JPA uses Coherence. These implementations reside in the `oracle.eclipselink.coherence.integrated` package. See "JPA on the Grid Configurations" on page 1-3 for more information.

- If you have existing Native ORM applications, then you can use the EclipseLink Native Object Relational Mapping (ORM) framework with them. The Native ORM approach is very similar to **JPA on the Grid**, however, it does not use annotations to configure how the cache is used. Instead, this approach employs an amendment method that defines the appropriate cache behavior. See "EclipseLink Native ORM Configurations" on page 1-19 for more information.
You can use the Coherence API with caches backed by TopLink Grid to access relational data with special cache loader and cache store interfaces which have been implemented for JPA.

In this traditional Coherence approach, TopLink Grid provides the CacheLoader and CacheStore implementations in the oracle.eclipselink.coherence.standalone package that are optimized for EclipseLink JPA. This technique is described in "Using TopLink Grid with Coherence Client Applications" on page 2-1.

When integrating JPA applications with the Coherence data grid, note the potential benefits and restrictions. You must understand how the grid works and how it relates to your JPA configurations to realize the full potential.

1.2 Required Files

The required files for working with TopLink Grid are the javax.persistence_2.0.0.0_2-0.jar, the eclipselink.jar, and the toplink-grid.jar. Assuming that you performed a standard installation of Coherence, these files can be found in the following locations:

- .../Oracle_Home/oracle_common/modules/javax.persistence_2.0.0.0_2-0.jar
- .../Oracle_Home/oracle_common/modules/oracle.toplink12.1.2/eclipselink.jar
- .../Oracle_Home/oracle_common/modules/oracle.toplink12.1.2/toplink-grid.jar

1.3 JPA on the Grid Configurations

This section describes JPA on the Grid and how to read and write objects in the Grid Cache, Grid Read, and Grid Entity configurations. It also describes how to work with queries against the Coherence cache under these configurations.

This section contains the following:

- Understanding JPA on the Grid
- JPA on the Grid API
- Grid Cache Configuration
- Grid Read Configuration
- Grid Entity Configuration
- Handling Grid Read and Grid Entity Failovers
- Wrapping and Unwrapping Entity Relationships
- Working with Queries

1.3.1 Understanding JPA on the Grid

The expression JPA on the Grid refers to using JPA and the power of the data grid to build applications with improved scalability and performance. In the JPA on the Grid approach, TopLink Grid provides a set of cache and query configuration options that allow you to control how EclipseLink JPA uses Coherence.

You can configure Coherence as a distributed shared (L2) cache or use Coherence as the primary data store. You can also configure entities to execute queries in the
Coherence data grid instead of the database. This allows clustered application deployments to scale beyond database-bound operations.

Figure 1–1 illustrates the relationship between an application, TopLink, Coherence, and the database.

![Figure 1–1 JPA on the Grid Approach](image)

1.3.2 **JPA on the Grid API**

The API used by JPA on the Grid configurations are shipped in the `toplink-grid.jar` file. Table 1–1 lists some of the key classes in the `oracle.eclipselink.coherence.integrated` package that are used in JPA on the Grid configurations.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>oracle.eclipseLink.coherence.integrated.EclipseLinkJPACacheLoader</code></td>
<td>Provides JPA-aware versions of the Coherence CacheLoader interface.</td>
</tr>
<tr>
<td><code>oracle.eclipseLink.coherence.integrated.EclipseLinkJPACacheStore</code></td>
<td>Provides JPA-aware versions of the Coherence CacheStore interface.</td>
</tr>
<tr>
<td><code>oracle.eclipseLink.coherence.integrated.config.CoherenceReadCustomizer</code></td>
<td>Enables a Coherence read configuration.</td>
</tr>
<tr>
<td><code>oracle.eclipseLink.coherence.integrated.config.CoherenceReadWriteCustomizer</code></td>
<td>Enables a Coherence read/write configuration.</td>
</tr>
<tr>
<td><code>oracle.eclipseLink.coherence.integrated.config.GridCacheCustomizer</code></td>
<td>Enables cache instances to be cached in Coherence instead of in the internal EclipseLink shared cache. All calls to the internal TopLink L2 cache are redirected to Coherence.</td>
</tr>
</tbody>
</table>
The configuration also uses the standard JPA run-time configuration file `persistence.xml` and the JPA mapping file `orm.xml`. You must also use the Coherence cache configuration file `coherence-cache-config.xml` to override the default Coherence settings and define the cache store caching scheme.

### 1.3.3 Grid Cache Configuration

The Grid Cache configuration can be considered as the base configuration for TopLink Grid. In this configuration, Coherence acts as the TopLink shared (L2) cache. This brings the power of the Coherence data grid to JPA applications that rely on database-hosted data that cannot be entirely preloaded into a Coherence cache. Some reasons why the data might not be able to be preloaded include extremely complex queries that exceed the abilities of Coherence Filters, third-party database updates that create stale caches, and reliance on native SQL queries, stored procedures, or triggers.

By using Coherence as the TopLink Grid cache, you can scale TopLink up into large clusters while avoiding the need to coordinate local shared caches. Updates made to entities are available in all Coherence cluster members immediately, upon committing a transaction.

In general, read and write operations in a Grid Cache configuration have the following characteristics:

- A primary key query will attempt to get entities first from the Coherence cache. If the attempt is unsuccessful, the database will be queried and the Coherence cache will be updated with the query results. See the following section, "Reading Objects in Grid Cache Configuration".

- A nonprimary key query will be executed against the database and the results will be checked against the Coherence cache. This is to avoid the negative performance impact of constructing entities that are already cached. Newly queried entities are put into the Coherence cache.

- A write operation will update the database and, if successfully committed, will put updated entities into the Coherence cache. See "Writing Objects in Grid Cache Configuration" on page 1-6.

See "Grid Cache Configuration Examples" on page 1-7 for detailed examples.

To use Coherence as a distributed cache for an entity, you must enable shared caching in EclipseLink. Shared caching is enabled by default for all entities, but the default can be explicitly set to `true` or `false` by setting the `eclipselink.cache.shared.default` property in the `persistence.xml` file. Specific entities can override the default using the `@Cache` annotation or by specifying the corresponding XML `<cache>` element in the `eclipselink-orm.xml` file. For more information, see:

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG)#How_to_Use_the_.40Cache_Annotation

#### 1.3.3.1 Reading Objects in Grid Cache Configuration

In the Grid Cache configuration, all read queries are directed to the database except primary key queries, which are directed to the Coherence cache first. Any cache misses will result in a database query.
All entities queried from the database are placed in the Coherence cache. This makes the entities immediately available to all members of the cluster. This is valuable because, by default, TopLink uses the cache to avoid constructing new entities from database results.

For each row resulting from a query, TopLink uses the primary key of the result row to query the corresponding entity from the cache. If the cache contains the entity then the entity is used and a new entity is not built. This approach can greatly improve application performance, especially with a warmed cache, because it reduces the cost of a query by eliminating the cost associated with object building.

Figure 1–2 illustrates the path of a read query in the Grid Cache configuration:

1. The application issues a find query.
2. For primary key queries, TopLink queries the Coherence cache first.
3. If the object does not exist in the Coherence cache, TopLink queries the database.
   - For all read queries except primary key queries, TopLink queries the database first.
4. Read objects are put into the Coherence cache.

**Figure 1–2 Reading Objects in Grid Cache Configuration**

1.3.3.2 Writing Objects in Grid Cache Configuration

In the Grid Cache configuration, TopLink performs all database write operations (insert, update, delete). The Coherence cache is then updated to reflect the changes made to the database. TopLink offers a number of performance features when writing large amounts of data including batch writing, parameter binding, stored procedure support, and statement ordering to ensure that database constraints are satisfied.

Figure 1–3 illustrates the path for writing and persisting objects in the Grid Cache configuration:
1. The application issues a commit query.

2. TopLink updates the database.

3. After a successful transaction, TopLink updates the Coherence cache.

**Figure 1–3  Writing and Persisting Objects in Grid Cache Configuration**

1.3.3.3 Grid Cache Configuration Examples

You can obtain the code in these examples at the following URL:


1.3.3.3.1 Configuring the Cache for the Grid Cache Configuration  The cache configuration file (coherence-cache-config.xml) in Example 1–1 defines the cache and configures a wrapper serializer to support serialization of relationships.

**Example 1–1 Configuring the Cache in Grid Cache Configuration**

```
<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>*</cache-name>
      <scheme-name>eclipselink-distributed</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <distributed-scheme>
      <scheme-name>eclipselink-distributed</scheme-name>
      <service-name>EclipseLinkJPA</service-name>
    </distributed-scheme>
    <!-- Configure a wrapper serializer to support serialization of relationships. -->
    <serializer>
```

Integrating TopLink Grid with Oracle Coherence  1-7
<class-name>oracle.eclipselink.coherence.integrated.cache.
WrapperSerializer</class-name>
</serializer>
</backing-map-scheme>
<local-scheme>
  <scheme-name>unlimited-backing-map</scheme-name>
</local-scheme>
</backing-map-scheme>
</caching-schemes>
</cache-config>

1.3.3.3.2 Configuring an Entity for the Grid Cache Configuration
To configure an entity to use Grid Cache, use the @Customizer annotation and the
GridCacheCustomizer class as shown in Example 1–2. This class intercepts all TopLink
calls to the internal TopLink Grid cache and redirects them to the Coherence cache.

Example 1–2 Configuring the Entity in Grid Cache Configuration

import oracle.eclipselink.coherence.integrated.config.GridCacheCustomizer;
import org.eclipse.persistence.annotations.Customizer;

@Entity
@Customizer(GridCacheCustomizer.class)
public class Employee {
...

1.3.3.3.3 Inserting Objects for the Grid Cache Configuration
In Example 1–3, TopLink performs the insert to create a new employee. Entities are persisted through the
EntityManager and placed in the database. After a successful transaction, the Coherence cache is updated.

Example 1–3 Inserting Objects in Grid Cache Configuration

EntityManagerFactory emf = Persistence.createEntityManagerFactory("employee-pu");

// Create an employee with an address and telephone number.
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Employee employee = createEmployee();
em.persist(employee);
em.getTransaction().commit();
em.close();

1.3.3.3.4 Querying Objects for the Grid Cache Configuration
In Example 1–4, the named JPQL query is directed to the database. Query results are resolved against the
Coherence cache to avoid the cost of building objects that have previously been cached.

Example 1–4 Querying Objects in Grid Cache Configuration

EntityManagerFactory emf = Persistence.createEntityManagerFactory("employee-pu");

EntityManager em = emf.createEntityManager();
List<Employee> employees = em.createQuery("select e from Employee e where e.lastName = :lastName").setParameter("lastName", "Smith").getResultList();

for (Employee employee : employees) {
    System.err.println(employee);
    for (PhoneNumber phone : employee.getPhoneNumbers()) {
        System.err.println("\t" + phone);
    }
}
emf.close();

### 1.3.4 Grid Read Configuration

Use the Grid Read configuration for entities that require fast access to large amounts of (fairly stable) data and write changes synchronously to the database. For these entities, cache warming would typically be used to populate the Coherence cache, but individual queries could be directed to the database if necessary.

In general, read and write operations in a Grid Read configuration have the following characteristics:

- Read operations get objects from the Coherence cache. Configuring a cache loader has no impact on JPQL queries. See the next section, "Reading Objects in Grid Read Configuration".

- Write operations update the database and, if successfully committed, updated entities are put into the Coherence cache. See "Writing Objects in Grid Read Configuration" on page 1-10.

See "Grid Read Configuration Examples" on page 1-11 for detailed examples.

#### 1.3.4.1 Reading Objects in Grid Read Configuration

In the Grid Read configuration, all primary key and non-primary key queries are directed to the Coherence cache. To reduce query processing time, TopLink Grid supports parallel processing of queries across the data grid. Coherence contains data already in object form, avoiding the performance impact of database communication and object construction.

With the Grid Read configuration, if Coherence does not contain the entity requested by the find(...) method, then null is returned. However, if a cache loader is configured for the entity's cache, Coherence will attempt to load the object from the database. This is true only for primary key queries.

Configuring a cache loader has no impact on JPQL queries translated to Coherence filters. When searching with a filter, Coherence will operate only on the set of entities in the caches; the database will not be queried. However, it is possible to direct a query, on a query-by-query basis, to the database instead of to Coherence by using the oracle.eclipselink.coherence.integrated.querying.IgnoreDefaultRedirector class, as shown in following example:

```java
query.setHint(QueryHints.QUERY_REDIRECTOR, new IgnoreDefaultRedirector());
```

Any objects retrieved by a database query will be added to the Coherence cache so that they are available for subsequent queries. Because this configuration resolves all queries for an entity through Coherence by default, the Coherence cache should be warmed with all of the data that is to be queried.

In the Grid Read configuration, projection queries (reports) that extract data from a single entity type will also be directed to Coherence. For example, the following JPQL
query will return the first and last names of all employees contained in the Coherence cache.

\[
\text{select } e.\text{firstName}, e.\text{lastName} \text{ from Employee } e
\]

This type of query is useful when the entire entity is not required, for example when populating a drop-down list in a user interface.

A cache store is not compatible with the Grid Read configuration because the EclipseLink JPA will perform all database updates and then propagate the updated objects into Coherence. If you use a cache store, Coherence will attempt to write the objects again.

For complete information on using EclipseLink JPA query hints, see "JPA Query Customization Extensions" in Java Persistence API (JPA) Extensions Reference for Oracle TopLink and "About JPA Query Hints" in Oracle Fusion Middleware Understanding Oracle TopLink.

Figure 1–4 illustrates the path for a query in the Grid Read configuration:

1. The application issues a JPQL query.
2. TopLink executes a Filter on the Coherence cache.
3. TopLink returns results from the Coherence cache only; the database is not queried.

**Figure 1–4 Reading Objects with a Query**

1.3.4.2 Writing Objects in Grid Read Configuration

In the Grid Read configuration, TopLink performs all database write operations (insert, update, delete) directly. The Coherence caches are then updated to reflect the
changes made to the database. TopLink offers a number of performance features when writing large amounts of data. These include batch writing, parameter binding, stored procedure support, and statement ordering to ensure that database constraints are satisfied.

This approach offers the best possibilities: database updates are performed efficiently and queries continue to be executed in parallel across the Coherence data grid, with the option of directing individual queries to the database.

Figure 1–5 illustrates the path for writing and persisting objects in the Grid Read configuration:

1. The application issues a commit query.
2. TopLink updates the database.
3. After a successful transaction, TopLink updates the Coherence cache.

Figure 1–5 Writing and Persisting Objects in Grid Read Configuration

1.3.4.3 Grid Read Configuration Examples
You can obtain the code in these examples at the following URL:

1.3.4.3.1 Configuring the Cache in Grid Read Configuration The cache configuration file (coherence-cache-config.xml) in Example 1–5 defines the cache and configures a wrapper serializer to support serialization of relationships. The oracle.eclipselink.coherence.integrated.EclipseLinkJPAECacheLoader class defines the cache store scheme.

Example 1–5 Configuring the Cache in Grid Read Configuration

<cache-config>
<caching-scheme-mapping>
  <cache-mapping>
    <cache-name>*</cache-name>
    <scheme-name>eclipselink-distributed-readonly</scheme-name>
  </cache-mapping>
</caching-scheme-mapping>
<caching-schemes>
  <distributed-scheme>
    <scheme-name>eclipselink-distributed-readonly</scheme-name>
    <service-name>EclipseLinkJPAReadOnly</service-name>
    <!--
    Configure a wrapper serializer to support serialization of relationships.
    -->
    <serializer>
      <class-name>oracle.eclipselink.coherence.integrated.cache.
      WrapperSerializer</class-name>
    </serializer>
    <backing-map-scheme>
      <read-write-backing-map-scheme>
        <internal-cache-scheme>
          <local-scheme />
        </internal-cache-scheme>
        <!--
        Define the cache scheme.
        -->
        <cachestore-scheme>
          <class-scheme>
            <class-name>oracle.eclipselink.coherence.integrated.
            EclipseLinkJPACacheLoader</class-name>
          </class-scheme>
          <init-params>
            <param-type>java.lang.String</param-type>
            <param-value>{cache-name}</param-value>
          </init-param>
          <init-param>
            <param-type>java.lang.String</param-type>
            <param-value>employee-pu</param-value>
          </init-param>
        </init-params>
        <read-only>true</readonly>
      </read-write-backing-map-scheme>
    </backing-map-scheme>
    <autostart>true</autostart>
  </distributed-scheme>
</caching-schemes>
</cache-config>

1.3.4.3.2 Reading Objects for the Grid Read Configuration

To configure an entity to read through a Coherence cache, use the @Customizer annotation and the
CoherenceReadCustomizer class as shown in Example 1–6:

**Example 1–6 Configuring the Entity in Grid Read Configuration**

```java
import oracle.eclipselink.coherence.integrated.config.CoherenceReadCustomizer;
import org.eclipse.persistence.annotations.Customizer;

@Entity
@Customizer(CoherenceReadCustomizer.class)
public class Employee {
```
1.3.4.3 Inserting Objects for the Grid Read Configuration  In Example 1–7, TopLink performs an insert to create a new employee. If the transaction is successful, the new object is placed into the Coherence cache under its primary key.

Example 1–7  Inserting Objects in Grid Read Configuration

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("employee-pu");
// Create an employee with an address and telephone number
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Employee employee = createEmployee();
em.persist(employee);
em.getTransaction().commit();
em.close();
emf.close();
```

1.3.4.4 Querying Objects for the Grid Read Configuration  When finding an employee, the read query is directed to the Coherence cache. The JPQL query is translated to Coherence filters, as shown in Example 1–8.

Example 1–8  Querying Objects in Grid Read Configuration

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("employee-pu");
EntityManager em = emf.createEntityManager();
List<Employee> employees = em.createQuery("select e from Employee e where e.lastName = :lastName").setParameter("lastName", "Smith").getResultList();
for (Employee employee : employees) {
    System.err.println(employee);
    for (PhoneNumber phone : employee.getPhoneNumbers()) {
        System.err.println("\t" + phone);
    }
}
emf.close();
```

To retrieve an object from the Coherence cache with a specific ID (key), use the `em.find(Entity.class, ID)` method. You can also configure a Coherence cache loader to query the database to find the object, if the cache does not contain the object with the specified ID.

1.3.5 Grid Entity Configuration

The Grid Entity configuration should be used by applications that require fast access to large amounts of (fairly stable) data, but perform relatively few updates. This configuration can be combined with a Coherence cache store using write-behind to improve application response time by performing database updates asynchronously.

In general, read and write operations in a Grid Entity configuration have the following characteristics:

- Read operations get objects from the Coherence cache. See "Reading Objects in Grid Entity Configuration" on page 1-14.
- Write operations put objects into the Coherence cache. If a cache store is configured, TopLink also performs write operations on the database. See "Writing Objects in Grid Entity Configuration" on page 1-14.
1.3.5.1 Reading Objects in Grid Entity Configuration
In the Grid Entity configuration, querying objects is identical to the Grid Read configuration. See "Reading Objects in Grid Cache Configuration" on page 1-5 for more information.

1.3.5.2 Writing Objects in Grid Entity Configuration
In the Grid Entity configuration, all objects that are persisted, updated, or merged through an EntityManager instance will be put in the appropriate Coherence cache. To persist objects in a Coherence cache to the database, an EclipseLink JPA cache store (oracle.eclipselink.coherence.integrated.EclipseLinkJPACacheStore) must be configured for each cache.

You can also configure the cache store to use write-behind to asynchronously batch-write updated objects. See Oracle Fusion Middleware Developing Applications with Oracle Coherence for more information.

Figure 1–6 illustrates the path for writing and persisting objects in the Grid Entity configuration.

1. The application issues a commit call.
2. TopLink directs all queries to update the Coherence cache.
3. By configuring a Coherence cache store (optional), TopLink will also update the database.

Figure 1–6 Writing and Persisting Objects in Grid Entity Configuration

1.3.5.3 Limitations on Writing Objects in Grid Entity Configuration
When using a cache store, Coherence assumes that all write operations succeed and will not inform TopLink of a failure. This could result in the Coherence cache differing
from the database. You cannot use optimistic locking to protect against data
corruption that may occur if the database is concurrently modified by Coherence and a
third-party application.

Because the order in which Coherence cache members write updates to the database is
unpredictable, referential integrity cannot be guaranteed. Referential integrity
constraints must be removed from the database. If they are not, write operations could
fail with the following error:

org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: java.sql.BatchUpdateException: ORA-02292: integrity constraint
violated - child record found
Error Code: 2292

1.3.5.4 Grid Entity Configuration Examples
You can obtain the code in these examples at the following URL:

http://www.oracle.
com/technetwork/middleware/toplink/examples-325517-en-ca.html

1.3.5.4.1 Configuring the Cache for the Grid Entity Configuration
The cache configuration
file (coherence-cache-config.xml) in Example 1–9 configures a wrapper serializer to
support serialization of relationships. The oracle.eclipselink.coherence.
integrated.EclipseLinkJPACacheStore class defines the cache store scheme.

Example 1–9 Configuring the Cache in Grid Entity Configuration

<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>*</cache-name>
      <scheme-name>eclipselink-distributed-readwrite</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <distributed-scheme>
      <scheme-name>eclipselink-distributed-readwrite</scheme-name>
      <service-name>EclipseLinkJPAReadWrite</service-name>
      <!--
      Configure a wrapper serializer to support serialization of relationships.
      -->
      <serializer>
        <class-name>oracle.eclipselink.coherence.integrated.cache.
WrapperSerializer</class-name>
      </serializer>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
1.3.5.4.2 Configuring an Entity for the Grid Entity Configuration

To configure an entity to read through Coherence, use the @Customizer annotation and the CoherenceReadWriteCustomizer class as shown Example 1–10:

Example 1–10 Configuring an Entity in Grid Entity Configuration

```java
import oracle.eclipselink.coherence.integrated.config.CohereceReadWriteCustomizer;
import org.eclipse.persistence.annotations.Customizer;

@Entity
@Customizer(CoherenceReadWriteCustomizer.class)
public class Employee {
    ...
}
```

1.3.5.4.3 Persisting Objects for the Grid Entity Configuration

In Example 1–11, TopLink performs the insert to create a new employee. Entities persist through the EntityManager instance and are placed in the appropriate Coherence cache.

Example 1–11 Persisting Objects in Grid Entity Configuration

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("employee-pu");

// Create an employee with an address and telephone number.
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Employee employee = createEmployee();
em.persist(employee);
em.getTransaction().commit();
em.close();
```

1.3.5.4.4 Querying Objects for the Grid Entity Configuration

When finding an employee, the read query is directed to the Coherence cache, as shown in Example 1–12.

Example 1–12 Querying Objects in Grid Entity Configuration

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("employee-pu");

EntityManager em = emf.createEntityManager();
List<Employee> employees = em.createQuery("select e from Employee e where e.lastName = :lastName").setParameter("lastName", "Smith").getResultList();

for (Employee employee : employees) {
    ...
}
```
System.err.println(employee);
for (PhoneNumber phone : employee.getPhoneNumbers()) {
    System.err.println("\t" + phone);
}
}
emf.close();

To get an object from the Coherence cache with a specific ID (key), use the em.find(Entity.class, ID) method. You can also configure a Coherence cache store to query the database to find the object, if the cache does not contain the object with the specified ID.

1.3.6 Handling Grid Read and Grid Entity Failovers

In the Grid Read and Grid Entity configurations, TopLink Grid will attempt to translate JPQL queries into Coherence Filters and execute the query in the grid. However some queries cannot be translated into filters. When TopLink Grid encounters such a query, it automatically fails over to the database to execute the query. In TopLink, you can specify a user-defined translation failure delegate object that will be called if the JPQL-to-filter translation fails. You configure the translation failure delegate by declaring the eclipselink.coherence.query.translation-failure-delegate persistence unit property. For example:

```xml
<property name="eclipselink.coherence.query.translation-failure-delegate" value="org.example.ExceptionFailoverPolicy"/>
```

A translation failure delegate must implement oracle.eclipselink.coherence.integrated.querying.TranslationFailureDelegate class which defines the single method translationFailed(DatabaseQuery query, Record arguments, Session session).

1.3.7 Wrapping and Unwrapping Entity Relationships

When storing entities with relationships in the Coherence cache, TopLink Grid generates a wrapper class that maintains the relationship information. In this way, when the object is read from the Coherence cache (eager or lazy), the relationships can be resolved.

If you read entities directly from the Coherence cache using the Coherence API, the wrappers are not automatically removed. You can configure automatic unwrapping programatically by calling the setNotEclipseLink(true) method on the serializer, as shown in Example 1–13. You can also set the system property as eclipselink.coherence.not-eclipselink to automatically unwrap an entity.

When configured properly, a cache get operation will return the unwrapped entity.

```
Example 1–13 Unwrapping an Entity

WrapperSerializer wrapperSerializer = (WrapperSerializer)myCache.getCacheService().getSerializer();
wrapperSerializer.setNotEclipseLink(true); // So the Serializer will unwrap an Entity when clients use a get() call from the cache.
```

1.3.8 Working with Queries

This section includes information on the following topics:

- Querying Objects by ID
1.3.8.1 Querying Objects by ID
To get an entity from the Coherence cache with a specific ID (key), use the `em.find(Entity.class, ID)` method. For example, the following code will get the entity with key 8, from the Coherence Employee cache.

```java
em.find(Employee.class, 8)
```

If the entity is not found in the Coherence cache, TopLink executes a `SELECT` statement against the database. If a result is found, then the entity is constructed and placed into the Coherence cache. The query’s specific behavior will depend on your Coherence cache configuration:

- calling the `find` method with a Grid Cache Configuration performs a `SELECT` statement against the database on a cache miss and then updates the cache.
- calling the `find` method with a Grid Read Configuration or a Grid Entity Configuration performs a `get` operation on the Coherence cache. A cache miss results in a `SELECT` statement against the database by using a CacheLoader instance, if it is configured.

1.3.8.2 Querying Objects with Criteria
To retrieve an entity that matches a specific selection criterion, use the `em.createQuery("...")` method. The query’s specific behavior will depend on your Coherence cache configuration:

- For the Grid Cache Configuration, the query will always execute a `SELECT` statement against the database. For example, the following code will execute a `SELECT` statement to find employees named John.

  ```java
  em.createQuery("select e from Employee e where e.name='John'")
  ```

- For the Grid Read Configuration and Grid Entity Configuration, the query will be executed against the Coherence cache. If the cache does not contain any entities that match the selection criteria, then nothing will be returned. This is an example of why the cache should be warmed before performing the query.

- For the cache store and cache loader, queries are performed only on primary keys

1.3.8.3 Using Indexes in Queries
Indexes allow values (or attributes of those values) and corresponding keys to be correlated within a cache to improve query performance. TopLink Grid allows you to declare indexes with the `@Property` annotation. The `IntegrationProperties` class provides the `INDEXED` property.

In Example 1–14, the `@Property` annotation declares that the name attribute is to be indexed. TopLink Grid will define an index for that attribute in the Publisher cache.

**Example 1–14 Exposing a Coherence Query Index to TopLink Grid**

```java
import static oracle.eclipselink.coherence.IntegrationProperties.INDEXED;
import oracle.eclipselink.coherence.integrated.config.CoherenceReadCustomizer;

@Customizer(CoherenceReadCustomizer.class)
```
public class Publisher implements Serializable {
    ...  
    @Property(name=INDEXED, value="true")
    private String name;
    ...

    With an index in place, you can issue a JPQL query, such as the following, to return all the Publishers in the cache with a name beginning with S.

    SELECT Publisher p WHERE p.name like 'S%'

    Internally, Coherence will process the query by consulting the name index to find matches rather than by deserializing and examining every Publisher object stored in the grid. By avoiding deserialization, you achieve a significant positive improvement on query execution time, eliminate garbage collection of the temporarily deserialized objects, and reduce CPU usage.

1.3.8.4 Limitations on Queries

The following are limitations on querying Coherence caches:

- Because the Coherence Filter framework is limited to a single cache, JPQL join queries cannot be translated to Filters. All join queries will execute on the database.

- This release of TopLink Grid does not provide support for JPQL bulk updates and deletions.

1.4 EclipseLink Native ORM Configurations

This section describes the EclipseLink Native Object Relational Mapping (ORM), an extensible object-relational mapping framework. It also describes how to configure amendment methods with Oracle JDeveloper and how to configure EclipseLink Native ORM cache stores and cache loaders.

This section contains the following:

- Understanding EclipseLink Native ORM
- API for EclipseLink Native ORM
- Configuring an Amendment Method
- Configuring the EclipseLink Native ORM Cache Store and Cache Loader

1.4.1 Understanding EclipseLink Native ORM

EclipseLink Native ORM provides an extensible object-relational mapping framework. It provides high-performance object persistence with extended capabilities configured declaratively through XML. These extended capabilities include caching (including support for clustered caching), advanced database-specific capabilities, and performance tuning and management options.

Like JPA on the Grid configurations, applications that employ EclipseLink ORM can access Coherence caches. However, unlike JPA on the Grid configurations, EclipseLink ORM applications do not use the @Customizer annotation to configure how the cache is used. Instead, they typically call an amendment method that defines the appropriate cache behavior.
1.4.2 API for EclipseLink Native ORM

The cache store and cache loader API used in EclipseLink Native ORM configurations are shipped in the toplink-grid.jar file. Table 1–2 describes the API for EclipseLink Native ORM. These classes can be found in the `oracle.eclipselink.coherence.integrated` package.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EclipseLinkNativeCacheStore(String cacheName,</td>
<td>Coherence cache store that should be used with native EclipseLink configuration (sessions.xml).</td>
</tr>
<tr>
<td>String sessionName)</td>
<td></td>
</tr>
<tr>
<td>EclipseLinkNativeCacheLoader(String cacheName,</td>
<td>Coherence cache loader that should be used with native EclipseLink configuration (sessions.xml).</td>
</tr>
<tr>
<td>String sessionName)</td>
<td></td>
</tr>
<tr>
<td>CoherenceReadCustomizer</td>
<td>Enables a Coherence read configuration.</td>
</tr>
<tr>
<td>CoherenceReadWriteCustomizer</td>
<td>Enables a Coherence read/write configuration.</td>
</tr>
<tr>
<td>GridCacheCustomizer</td>
<td>Enables entity instances to be cached in Coherence instead of in the internal EclipseLink shared cache</td>
</tr>
</tbody>
</table>

Note that the second initialization parameter in the signatures, `sessionName`, represents the name of the mapping project that must be listed in the native EclipseLink configuration file, `META-INF/sessions.xml`.

The `EclipseLinkNativeCacheStore` and `EclipseLinkNativeCacheLoader` classes allow applications that use EclipseLink Native ORM to access Coherence caches. Use these classes when Coherence cache behavior has been configured through an amendment method. These classes can be used to configure a cache store or cache loader for each persistent class in the same way as described in "JPA on the Grid Configurations" on page 1-3.

Use the Coherence cache configuration file `coherence-cache-config.xml` to define the cache store caching scheme and to override any default Coherence settings.

The configuration uses the native EclipseLink sessions.xml file and the project.xml file. The sessions.xml file, and all of the deployment XML files (which have user-defined names) listed in it, must be available on the classpath or packaged within a JAR file within the `META-INF` directory.

You must also configure an amendment method to define the appropriate cache behavior. See "Configuring an Amendment Method" for more information.

1.4.3 Configuring an Amendment Method

An amendment method is a method that uses the EclipseLink descriptor API to customize the ORM mapping metadata for a class. The method is called when the descriptor is loaded at runtime. The purpose of the amendment methods provided by TopLink Grid is to define how the Coherence cache is going to be used. Amendment methods are the TopLink native ORM alternative to the @Customizer annotation; they produce the same configuration.

The TopLink Grid customizer classes in the toplink-grid.jar file (CoherenceReadCustomizer, CoherenceReadWriteCustomizer, and GridCacheCustomizer) provide an afterLoad amendment method that can be selected to enable the appropriate Coherence cache behavior.
You can select the amendment method using either JDeveloper or EclipseLink Workbench. The following section describes how to configure the amendment method with JDeveloper. A description of EclipseLink Workbench is beyond the scope of this document.

### 1.4.3.1 Configuring the Amendment Method in JDeveloper

To configure an amendment method:

1. In the JDeveloper Structure pane, expand the desired `tlMap` descriptor name.

![Figure 1–7  `tlMap` Descriptors in the JDeveloper Structure Pane](image)

2. Right-click the desired TopLink descriptor element. Select **Advanced Properties** to open the **Advanced Properties** dialog box. Select the **After Loading** check box and click **OK**.
3. In the After Load tab of the tlMap configuration window, enter the name of the class containing the afterLoad amendment method you want to use for the selected TopLink descriptor. You can also use the class browser to search for the class. Figure 1–9 illustrates the After Load tab of the tlMap configuration window.
Once the descriptor is loaded at run time, the following static method will be executed. The method must have the following characteristics: (1) be public static and (2) take a single parameter of type `oracle.toplink.descriptors.ClassDescriptor`.

Figure 1–10 illustrates the class browser with the `CoherenceReadCustomizer` class selected.

Figure 1–10  Searching for the Class containing the Amendment Method
4. In the After Load tab of the tlMap configuration window, select the amendment method from the Static Method dropdown list. For the Coherence Customizer classes, this will be the afterLoad method.

**Figure 1–11 Selecting the Amendment Method**

![tlMap configuration window](image)

Once the descriptor is loaded at runtime, the following static method will be executed. The method must have the following characteristics: (1) be public static and (2) take a single parameter of type oracle.toplink.descriptors.ClassDescriptor.

- **Class**: `oracle.eclipselink.coherence.integrated.config.CoherenceReadCustomizer`
- **Static Method**: `afterLoad(ClassDescriptor): void`

### 1.4.4 Configuring the EclipseLink Native ORM Cache Store and Cache Loader

The `coherence-cache-config.xml` file must specify the cache loader or cache store class and provide parameters for the cache name and session name (that is, project name). The following examples illustrate that aside from changing the class name (`EclipseLinkNativeCacheStore` or `EclipseLinkNativeCacheLoader`), you do not have to make any changes to the Coherence cache configuration depending on whether you are using the cache loader or cache store.

**Example 1–15** illustrates a configuration in the `coherence-cache-config.xml` file for a cache that can communicate with EclipseLink Native ORM applications. The `class-name` element identifies the `EclipseLinkNativeCacheStore` class as the cache store scheme. The `param-value` elements specify the cache name and the session (project) name that are passed to the class.

**Example 1–15  Configuration for an Integrated EclipseLinkNativeCacheStore**

```xml
...<distributed-scheme>
    <scheme-name>eclipselink-native-distributed-store</scheme-name>
    <service-name>EclipseLinkNative</service-name>
    <serializer>
        <class-name>oracle.eclipselink.coherence.integrated.cache.WrapperSerializer</class-name>
    </serializer>
    <backing-map-scheme>
        <read-write-backing-map-scheme>
            <internal-cache-scheme></internal-cache-scheme>
        </read-write-backing-map-scheme>
    </backing-map-scheme>
</distributed-scheme>
```

EclipseLinkNativeCacheStore
Using POF Serialization with TopLink Grid and Coherence

Example 1–16 illustrates an integrated EclipseLinkNativeCacheLoader instance configuration in the coherence-cache-config.xml file. The cache name (\{cache-name\}) and session name (coherence-native-project) parameter values are passed to the class.

**Example 1–16 Configuration for an Integrated EclipseLinkNativeCacheLoader**

...<cachestore-scheme>
   <class-scheme>
      <class-name>oracle.eclipselink.coherence.integrated.EclipseLinkNativeCacheLoader</class-name>
      <init-params>
         <init-param>
            <param-type>java.lang.String</param-type>
            <param-value>{cache-name}</param-value>
         </init-param>
         <init-param>
            <param-type>java.lang.String</param-type>
            <param-value>coherence-native-project</param-value>
         </init-param>
      </init-params>
   </class-scheme>
</cachestore-scheme>

1.5 Using POF Serialization with TopLink Grid and Coherence

This section describes how to use Portable Object Format (POF) serialization to optimize the performance of applications that use TopLink Grid and Coherence caches.

Serialization is the process of encoding an object into a binary format. It is a critical component when working with Coherence as data must be moved around the network. The Portable Object Format (also referred to as POF) is a language agnostic binary format. POF was designed to be incredibly efficient in both space and time and has become a cornerstone element in working with Coherence. Using POF has many advantages ranging from performance benefits to language independence. It's recommended that you look closely at POF as your serialization solution when working with Coherence.
This section focuses only on the changes and additions that you need to make to your TopLink application files to make them eligible to participate in POF serialization. For more detailed information on using and configuring POF, see "Using Portable Object Format" in the *Oracle Fusion Middleware Developing Applications with Oracle Coherence*.

This section contains the following:

- Implement a Serialization Routine
- Define a Cache Configuration File
- Define a POF Configuration File

### 1.5.1 Implement a Serialization Routine

You must implement serialization routines that know how to serialize and deserialize your Entities. You can do this by implementing the `PortableObject` interface or by creating a serializer using the `com.tangosol.io.pof.PofSerializer` interface.

- Implement the `PortableObject` interface in your Entity class files
  
  The `com.tangosol.io.pof.PortableObject` interface provides classes with the ability to self-serialize and deserialize their state to and from a POF data stream. To use this interface, you must also provide implementations of the required methods `readExternal` and `writeExternal`.

  Example 1–17 illustrates a sample Entity class file that implements the `PortableObject` interface. Note the implementations of the required `readExternal` and `writeExternal` methods.

  Also note that the class includes an `@OneToOne` annotation to define the relationship mapping between the `Trade` object and a `Security` object. TopLink supports all of the relationship mappings defined by the JPA specification: one-to-one, one-to-many, many-to-many, and many-to-many. These relationships can be expressed as annotations.

**Example 1–17  Sample Entity Class that Implements PortableObject**

```java
package oracle.toplinkgrid.codesample.pof.models.trader;

import java.io.IOException;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import com.tangosol.io.pof.PofReader;
import com.tangosol.io.pof.PofWriter;
import com.tangosol.io.pof.PortableObject;

/**
 * This class will not be stored within Coherence as Trades are not high
 * throughput objects in this model.
 * /
 * @Entity
 * public class Trade implements Serializable, PortableObject{
     /**
     * */
```
private static final long serialVersionUID = -244532585419336780L;

@Id
@GeneratedValue
protected long id;

@OneToOne(fetch=FetchType.EAGER)
protected Security security;

protected int quantity;
protected double amount;

public long getId() {
    return id;
}
public void setId(long id) {
    this.id = id;
}

public Security getSecurity() {
    return security;
}
public void setSecurity(Security security) {
    this.security = security;
}

public int getQuantity() {
    return quantity;
}
public void setQuantity(int quantity) {
    this.quantity = quantity;
}

public double getAmount() {
    return amount;
}
public void setAmount(double amount) {
    this.amount = amount;
}

public void readExternal(PofReader pofreader) throws IOException {
    id = pofreader.readLong(0);
    quantity = pofreader.readInt(2);
    amount = pofreader.readDouble(3);
}

public void writeExternal(PofWriter pofwriter) throws IOException {
    pofwriter.writeLong(0, id);
    pofwriter.writeInt(2, quantity);
    pofwriter.writeDouble(3, amount);
}

Create a POFSerializer for the Entities

An alternative to implementing the PortableObject interface is to implement the com.tangosol.io.pof.PofSerializer interface to create your own serializer and deserializer. This interface provides you with a way to externalize your serialization logic from the Entities you want to serialize. This is particularly useful when you do not want to change the structure of your classes to work with POF and Coherence. The POFSerializer interface provides these methods:

- public Object deserialize(PofReader in)
- public void serialize(PofWriter out, Object o)
1.5.2 Define a Cache Configuration File

In the cache configuration file, create cache mappings corresponding to the Entities you will be working with. Identify the serializer (such as `com.tangosol.io.pof.ConfigurablePofContext`) and the POF configuration file `pof-config.xml`. Identify the EclipseLink cache store (such as `oracle.eclipselink.coherence.integrated.EclipseLinkJPACacheStore`) in the `<cachestore-scheme>` attribute.

Example 1–18 Sample Cache Configuration File

```xml
<?xml version="1.0"?>
<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
    <caching-schemes>
        <distributed-scheme>
            <scheme-name>eclipselink-jpa-distributed-load</scheme-name>
            <service-name>EclipseLinkJPA</service-name>
            <serializer>
                <instance>
                    <class-name>com.tangosol.io.pof.ConfigurablePofContext</class-name>
                    <init-params>
                        <init-param>
                            <param-type>String</param-type>
                            <param-value>trader-pof-config.xml</param-value>
                        </init-param>
                    </init-params>
                </instance>
            </serializer>
        </distributed-scheme>
        <distributed-scheme>
            <scheme-name>eclipselink-jpa-distributed</scheme-name>
            <service-name>EclipseLinkJPA</service-name>
            <serializer>
                <instance>
                    <class-name>com.tangosol.io.pof.ConfigurablePofContext</class-name>
                    <init-params>
                        <init-param>
                            <param-type>String</param-type>
                            <param-value>trader-pof-config.xml</param-value>
                        </init-param>
                    </init-params>
                </instance>
            </serializer>
        </distributed-scheme>
    </caching-schemes>
    <autostart>true</autostart>
</cache-config>
```

... additional cache mappings...
...
1.5.3 Define a POF Configuration File

Provide a file that identifies the Entity classes that will participate in POF serialization. Coherence provides a POF configuration file which is named \texttt{pof-config.xml} by default. Use the file to assign type-ids of TopLink classes to the Entity classes.

TopLink Grid simplifies the assignment of type-ids to TopLink-Grid required classes. If the allow-interfaces element is set to true in the POF configuration file, then only one type-id entry is needed for TopLink-Grid classes.

- \texttt{oracle.eclipselink.coherence.integrated.cache}\texttt{.TopLinkGridPortableObject} — the TopLink Grid analog of the \texttt{PortableObject} interface. TopLink Grid classes that implement the \texttt{TopLinkGridPortableObject} interface can be POF serialized by the \texttt{TopLinkGridSerializer} class. This allows you to register a single class for all implementors of this interface when the allow-interfaces POF configuration element is set to true.

- \texttt{oracle.eclipselink.coherence.integrated.cache}\texttt{.TopLinkGridSerializer} — the associated serializer for all implementors of the \texttt{TopLinkGridPortableObject} interface. This allows you to register a single class for all implementors of this interface in your POF configuration XML file when the allow-interfaces POF configuration element is set to true.
Example 1–19 illustrates the assignment of the TopLinkGridPortableObject and TopLinkGridSerializer serializer class to the Attorney Entity.

Example 1–19  Simplified POF Configuration File

```xml
<?xml version="1.0"?>
<pof-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.oracle.com/coherence/coherence-pof-config"
    xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-pof-config/1.0/coherence-pof-config.xsd">
    <user-type-list>
        <!-- include all 'standard' Coherence POF user types -->
        <include>coherence-pof-config.xml</include>
        <user-type>
            <type-id>1163</type-id>
            <class-name>oracle.toplinkgrid.codesample.pof.models.trader.Attorney</class-name>
            <user-type>
                ... additional type IDs for Entity classes ...
                <user-type>
                    <type-id>1130</type-id>
                    <class-name>oracle.eclipselink.coherence.integrated.cache.TopLinkGridPortableObject</class-name>
                    <serializer>
                        ...</serializer>
                    </user-type>
                </user-type>
            </user-type>
        </user-type-list>
        <allow-interfaces>true</allow-interfaces>
    </pof-config>
```

Note: the allow-subclasses element is not required for a TopLink Grid POF configuration.

If you cannot set allow-interfaces to true, then you must define individual type-id entries for the following classes:

- oracle.eclipselink.coherence.integrated.internal.cache.ElementCollectionUpdateProcessor—Entry processor used by TopLink Grid to update an ElementCollection object within the cache.
- oracle.eclipselink.coherence.integrated.internal.cache.RelationshipUpdateProcessor—An internal file, used to update lazy-loaded relationship data into the grid.
- oracle.eclipselink.coherence.integrated.internal.cache.VersionPutProcessor—An internal file, used for optimistic lock-aware updates to the grid.
Using POF Serialization with TopLink Grid and Coherence

- **oracle.eclipselink.coherence.integrated.internal.cache.
  VersionRemoveProcessor**—An internal file, used for optimistic lock-aware removals from the grid.

- **oracle.eclipselink.coherence.integrated.internal.cache.
  SerializableWrapper**—A generic wrapper class for non POF serialization. It provides for serialization to a node which may not have the correct dynamic wrapper defined which would otherwise result in an exception.

- **oracle.eclipselink.coherence.integrated.internal.cache.
  LockVersionExtractor**—Used during conditional puts of Optimistically Locked objects. This class is used to extract the version value from the object.

- **oracle.eclipselink.coherence.integrated.internal.querying.
  FilterExtractor**—used by the filters to extract values from the objects stored in the caches. It supports both attribute access and method access.

- **oracle.eclipselink.coherence.integrated.internal.querying.
  EclipseLinkFilterFactory$SubClassOf**—An inner class. This is a Filter extension that filters on the type of Entity, eliminating superclasses from polymorphic queries. A type-id is needed for this class needed only if you are using this operation.

- **oracle.eclipselink.coherence.integrated.internal.querying.
  EclipseLinkFilterFactory$$IsNull**—An inner class. IsNull is equivalent to the Coherence IsNullFilter except that it provides support for a ValueExtractor instead of an explicit method name. A type-id is needed for this class needed only if you are using this operation.

- **oracle.eclipselink.coherence.integrated.internal.querying.
  EclipseLinkFilterFactory$$IsNotNull**—An inner class. IsNotNull is equivalent to the Coherence IsNotNullFilter except that it provides support for a ValueExtractor instead of an explicit method name. A type-id is needed for this class needed only if you are using this operation.

Example 1–20 illustrates a sample POF configuration file that includes definitions for the TopLink Grid support files.

**Example 1–20 Sample POF Configuration File with Definitions for TopLink Classes**

```xml
... additional type IDs for Entity classes ...
<user-type>
  <type-id>1144</type-id>
  <class-name>oracle.eclipselink.coherence.integrated.internal.cache.
  ElementCollectionUpdateProcessor</class-name>
</user-type>
<user-type>
  <type-id>1143</type-id>
  <class-name>oracle.eclipselink.coherence.integrated.internal.querying.
  FilterExtractor</class-name>
</user-type>
<user-type>
  <type-id>1142</type-id>
  <class-name>oracle.eclipselink.coherence.integrated.internal.cache.
  LockVersionExtractor</class-name>
</user-type>
```

Integrating TopLink Grid with Oracle Coherence 1-31
1.6 Best Practices

This section contains best practice recommendations on how to use TopLink Grid with byte code weaving, lazy loading, near caches, and cache configurations.

This section contains the following:

- Changing Compiled Java Classes with Byte Code Weaving
- Deferring Database Queries with Lazy Loading
- Defining Near Caches for Applications Using TopLink Grid
- Ensuring Prefixed Cache Names Use Wildcard in Cache Configuration
- Overriding the Default Cache Name

1.6.1 Changing Compiled Java Classes with Byte Code Weaving

Byte code weaving is a technique for changing the byte code of compiled Java classes. You can configure byte code weaving to enable a number of EclipseLink JPA performance optimizations, including support for the lazy loading of one-to-one and many-to-one relationships, attribute-level change tracking, and fetch groups.
Weaving can be performed either dynamically when entity classes are loaded, or statically as part of the build process. Static byte code weaving can be incorporated into an Ant build using the `weaver` task provided by EclipseLink.

Dynamic byte code weaving is automatically enabled in Java EE 5-compliant application servers such as Oracle WebLogic. However, in Java SE it must be explicitly enabled by using the JRE 1.5 `javaagent` JVM command line argument. See “Using Weaving” in Solutions Guide for Oracle TopLink.

To enable byte code weaving in a Coherence cache server, the Java VM should be invoked with `-javaagent:<PATH>/eclipselink.jar`. Java SE client applications should be run with the `-javaagent` argument.

See "Using Weaving" in Solutions Guide for Oracle TopLink for more information on configuring and disabling static and dynamic byte code weaving.

### 1.6.2 Deferring Database Queries with Lazy Loading

Lazy loading is a technique used to defer the querying of objects from the database until they are required. This can reduce the amount of data loaded by an application and improve throughput. A TopLink Grid JPA or native ORM application should lazily load all relationships. Lazy loading is the default for one-to-many and many-to-many relationships in JPA, but is eager for one-to-one and many-to-one relationships. You must explicitly select lazy loading on these relationship types. For example, you can specify lazy loading as an attribute for many of the relationship annotations:

```java
... @ManyToOne(fetch=FetchType.LAZY) private Publisher parent ... 
```

For maximum efficiency, lazy loading should be specified for all one-to-one and many-to-one entity relationships that TopLink Grid stores in the Coherence cache. Lazy loading is implemented through byte code weaving in EclipseLink and must be enabled explicitly if not running in a Java EE 5-compliant application server. For more information, see "Changing Compiled Java Classes with Byte Code Weaving" on page 1-32.

### 1.6.3 Defining Near Caches for Applications Using TopLink Grid

Near cache is one of the standard cache configurations offered by Oracle Coherence. The use of near caches can improve throughput by avoiding network access when an object is retrieved repeatedly. For example, in an environment where users are pinned to a particular Web server, near caching may improve performance.

The near cache is a hybrid cache consisting of a front cache, which is of limited size and offers fast data access, and a larger back cache, which can be scalable, can load on demand, and provide failover protection.

For applications using TopLink Grid, you configure the near cache in the same way as any other application using Oracle Coherence. See "Near Cache" and "Defining Near Cache Schemes" in the Oracle Fusion Middleware Developing Applications with Oracle Coherence for more information on near caches.
1.6.4 Ensuring Prefixed Cache Names Use Wildcard in Cache Configuration

When using TopLink Grid with applications that use Coherence caches and Coherence*Web, you might want to apply different configuration properties to the TopLink Grid caches for entities and the Coherence*Web caches. The most efficient way to specify and configure a set of caches is to use a wildcard character ("*"). However, this will match both sets of caches. To separate the Coherence*Web caches from entity caches, you must create a wildcard pattern that will match entities only. One way to do this is to prepend a unique prefix to the entity cache names.

The following steps describe how to create and use a custom session customizer to prepend a specified prefix to TopLink Grid-enabled classes.

1. Create a session customizer class that will prepend TopLink-enabled classes with a specified prefix.

Example 1–21 illustrates a custom session customizer class, CacheNamePrefixCustomizer, which implements the EclipseLink SessionCustomizer class. The class defines a PREFIX_PROPERTY myapp.cache-prefix that represents the prefix that will be added to the TopLink-enabled classes. The value of the property can be either specified in the persistence.xml file (described in Step 2) or passed in an optional property map to the Persistence.createEntityManagerFactory method.

Example 1–21 Session Customizer to Prepend

```java
import java.util.Collection;
import oracle.eclipselink.coherence.IntegrationProperties;
import oracle.eclipselink.coherence.integrated.cache.CoherenceInterceptor;
import oracle.eclipselink.coherence.integrated.internal.cache.CoherenceCacheHelper;
import org.eclipse.persistence.config.SessionCustomizer;
import org.eclipse.persistence.descriptors.ClassDescriptor;
import org.eclipse.persistence.sessions.Session;

public class CacheNamePrefixCustomizer implements SessionCustomizer {

    private static final String PREFIX_PROPERTY = "myapp.cache-prefix";

    public void customize(Session session) throws Exception {
        // Look up custom persistence unit cache prefix property
        String prefix = (String) session.getProperty(PREFIX_PROPERTY);
        if (prefix == null) {
            throw new RuntimeException("Cache name prefix customizer configured but prefix property " +
```
PREFIX_PROPERTY + "' not specified";

// Iterate over all entity descriptors
Collection<ClassDescriptor> descriptors = session.getDescriptors().values();
for (ClassDescriptor classDescriptor : descriptors)
{
    // If entity is TopLink Grid-enabled, prepend cache name with prefix
    if (CoherenceInterceptor.class.equals(classDescriptor.getClass()))
    {
        String cacheName = CoherenceCacheHelper.getCacheName(classDescriptor);
        classDescriptor.setProperty(IntegrationProperties.COHERENCE_CACHE_NAME, prefix + cacheName);
    }
}
}

2. Edit the persistence.xml file to declare a value for the prefix property.
   In the following example, MyApp_ is defined as the value of the prefix property myapp.cache-prefix in the persistence.xml file. The myapp.cache-prefix prefix property is defined in the custom session customizer file.

   <property name="myapp.cache-prefix" value="MyApp_"/>

   See http://www.eclipse.org/eclipselink/ for more information on the EclipseLink SessionCustomizer class.

3. Edit the persistence.xml file to add the name of the custom session customizer class as the value of the eclipselink.session.customizer context property.

   <property name="eclipselink.session.customizer" value="CacheNamePrefixCustomizer"/>

4. Edit the coherence-cache-config.xml file to add the name of the prefix with a wildcard character to the cache mapping.

   <cache-mapping>
     <cache-name>MyApp_*</cache-name>
     <scheme-name>eclipselink-distributed-readonly</scheme-name>
   </cache-mapping>

1.6.5 Overriding the Default Cache Name

There may be situations where you want to override the default name given to an entity cache. In TopLink Grid, entity cache names default to the entity name. The following list describes how the name of the cache can be determined, and how you can change it explicitly:

Cache Name—the cache name can be set either by default, or set explicitly:

- Default: cache name defaults to entity name. The entity name, in turn can be set either by default, or set explicitly:
  - Default: Entity name defaults to class short name.
  - Explicit: Entity name can be set explicitly by using the name property of the @Entity annotation.
- Explicit: the cache name can be set explicitly by using the @Property annotation.

For example, the following code fragment illustrates the Employee class. By default, the entity cache name would be Employee. However, you can force the name of the Employee entity cache to be EMP_CACHE by using the @Property annotation.
import static oracle.eclipselink.coherence.IntegrationProperties.COHERENCE_CACHE_NAME;
import org.eclipse.persistence.annotations.Property;

...
@Entity(name='Emp')
@Property(name=COHERENCE_CACHE_NAME, value="EMP_CACHE")
public class Employee implements Serializable {
...

Notice that the code explicitly specifies the entity name as Emp. If the name="Emp" value were not present, then the entity name would have defaulted to the short class name Employee.
This chapter describes how to use the Coherence API with caches backed by TopLink Grid to access relational data. It also describes how to use the native, entity-based Coherence implementations of the cache store and cache loader to access relational data. These implementations use JPA to load and store objects to the database.

**Note:** Only resource-local and bootstrapped entity managers can be used with Coherence API and JPA. Container-managed entity managers and those that use Java Transaction Architecture (JTA) transactions are not currently supported.

This chapter contains the following sections:
- Using TopLink Grid with Coherence Client Applications
- Using Third Party JPA Providers

### 2.1 Using TopLink Grid with Coherence Client Applications

This section describes how to use the Coherence API with caches backed by TopLink Grid to access relational data. Access to the relational data is provided with JPA cache loader and cache store interfaces which have been optimized for EclipseLink JPA.

In this traditional Coherence approach, TopLink Grid provides the `CacheLoader` and `CacheStore` implementations in the `oracle.eclipselink.coherence.standalone` package that are optimized for EclipseLink JPA.

**Figure 2–1** illustrates the relationship between the client application (which employs Coherence APIs), the Coherence cache, TopLink Grid, and the database.
2.1.1 API for Coherence with TopLink Grid Configurations

TopLink Grid uses the standard JPA run-time configuration file `persistence.xml` and the JPA mapping file `orm.xml`. The Coherence cache configuration file `coherence-cache-config.xml` must be specified to override the default Coherence settings and to define the cache store caching scheme.

The TopLink Grid cache store and cache loader implementations are shipped in the `toplink-grid.jar` file. The JAR file in installed with the Coherence product in the `...\oracle_common\modules\oracle.toplink_12.1.2` folder.

The TopLink Grid cache store and cache loader classes which are optimized for EclipseLink JPA and designed for use by Coherence applications, are in the `oracle.eclispelink.coherence.standalone` package. Table 2–1 describes these classes.

**Table 2–1** TopLink Grid Classes to build Coherence with TopLink Grid Applications

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EclipseLinkJPACacheLoader</td>
<td>Provides JPA-aware versions of the Coherence CacheLoader class.</td>
</tr>
<tr>
<td>EclipseLinkJPACacheStore</td>
<td>Provides JPA-aware versions of the Coherence CacheStore class.</td>
</tr>
</tbody>
</table>

2.1.2 Sample Cache Configuration File for Coherence with TopLink Grid

In the cache configuration file (`coherence-cache-config.xml`), define the cache as illustrated in Example 2–1. For TopLink Grid, you have to define only two parameters:

- The *name of the cache for the entity being stored*. Unless explicitly overridden in JPA this is the entity name that, by default, is the unqualified name of the entity class. In Example 2–1, the name of the cache is `Employee`. You can use the built-in Coherence macro `{cache-name}` to supply the name of the cache that is constructing and using the cache store.
The name of the persistence unit containing the entity being stored. In Example 2–1, employee-pu is a persistence unit defined in the META-INF/persistence.xml file that includes the Employee entity.

To define more entity caches, add additional <cache-mapping> elements.

Example 2–1 Configuring the Cache for Coherence with TopLink Grid

```xml
<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>Employee</cache-name>
      <scheme-name>distributed-eclipselink</scheme-name>
    </caching-scheme-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <distributed-scheme>
      <scheme-name>distributed-eclipselink</scheme-name>
      <service-name>EclipseLinkJPA</service-name>
      <backing-map-scheme>
        <read-write-backing-map-scheme>
          <internal-cache-scheme>
            <local-scheme />
          </internal-cache-scheme>
          <!-- Define the cache scheme. -->
          <cachestore-scheme>
            <class-scheme>
              <!-- Because the client code is using Coherence API, use the "standalone" version of the cache loader. -->
              <class-name>oracle.eclipselink.coherence.standalone.EclipseLinkJPACacheStore</class-name>
              <init-params>
                <!-- This parameter is the name of the cache containing the entity. -->
                <init-param>
                  <param-type>java.lang.String</param-type>
                  <param-value>{cache-name}</param-value>
                </init-param>
                <!-- This parameter is the persistence unit name. -->
                <init-param>
                  <param-type>java.lang.String</param-type>
                  <param-value>employee-pu</param-value>
                </init-param>
              </init-params>
            </class-scheme>
          </cachestore-scheme>
        </read-write-backing-map-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
```
2.1.3 Sample Project for Using Coherence with TopLink Grid

"Using JPA with Coherence" in the *Oracle Fusion Middleware Tutorial for Oracle Coherence* provides a sample project that uses the TopLink Grid cache store and cache loader classes which are optimized for EclipseLink JPA and designed for use by Coherence applications. These classes can be found in the oracle.eclipselink.coherence.standalone package.

The project uses the Oracle Express Database and the Eclipse IDE to configure a project for JPA, create the JPA persistence unit and entities, edit the persistence.xml file, create a cache configuration file for JPA, automatically generate JPA objects for a database table, and create a class to interact with the data objects.

2.2 Using Third Party JPA Providers

Oracle Coherence provides its own implementations of the CacheLoader and CacheStore classes which can be used with JPA. The JpaCacheLoader and JpaCacheStore classes do not have to use EclipseLink JPA—they can use any JPA implementation to load and store entities to and from a data store. The entities must be mapped to the data store and a JPA persistence unit configuration must exist. A JPA persistence unit is defined as a logical grouping of user-defined entity classes that can be persisted and their settings.

Coherence also provides a default cache configuration file named coherence-cache-config.xml. The JPA run-time configuration file, persistence.xml, and the default JPA Object-Relational mapping file, orm.xml, are typically provided by the JPA implementation.

2.2.1 API for Native Coherence JPA CacheStore and CacheLoader

The JpaCacheLoader and JpaCacheStore classes can be found in the coherence-jpa.jar file, which is installed in the ...\coherence\lib folder in the Coherence installation. The CacheLoader and CacheStore interfaces can be found in the coherence.jar file, which is also installed in the ...\coherence\lib folder.

Table 2–2 describes the default JPA implementations provided by Coherence.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.tangosol.coherence.jpa.JpaCacheLoader</td>
<td>The JPA implementation of the Coherence CacheLoader interface. Use this class as a load-only implementation. It can use any JPA implementation to load entities from a data store. The entities must be mapped to the data store and a JPA persistence unit configuration must exist. Use the JpaCacheStore class for a full load and store implementation.</td>
</tr>
<tr>
<td>com.tangosol.coherence.jpa.JpaCacheStore</td>
<td>The JPA implementation of the Coherence CacheStore interface. Use this class as a full load and store implementation. It can use any JPA implementation to load and store entities to and from a data store. The entities must be mapped to the data store and a JPA persistence unit configuration must exist. <strong>Note:</strong> The persistence unit is assumed to be set to use RESOURCE_LOCAL transactions.</td>
</tr>
</tbody>
</table>

---

2-4 Oracle Fusion Middleware Integrating Oracle Coherence
2.2.2 Steps to Use a Third Party JPA Provider and Native Coherence JPA API

To use a third party JPA provider and the native Coherence JPA API to load and store objects to the database:

1. **Obtain a JPA Provider Implementation.** The provider implementation allows you to map, query, and store Java objects to a database.

2. **Configure a Coherence JPA Cache Store.** The JPA cache store configuration maps database entities to Java objects.

2.2.2.1 Obtain a JPA Provider Implementation

A JPA provider allows you to work directly with Java objects, rather than with SQL statements. You can map, store, update and retrieve data, and the provider will perform the translation between database entities and Java objects.

The Coherence JPA cache store and cache loader work with any JPA-compliant implementation. Oracle recommends using EclipseLink JPA, the reference implementation for the JPA 2.0 specification. Oracle TopLink and TopLink Grid for Coherence integration include EclipseLink as their JPA implementations.

The TopLink Grid and EclipseLink JAR files (toplink-grid.jar and eclipselink.jar) are included in the Coherence installation and can be found in the $\ldots$\oracle_common\modules\oracle.toplink_12.1.2 folder.

2.2.2.2 Configure a Coherence JPA Cache Store

JPA is a standard API for mapping, querying, and storing Java objects to a database. The characteristics of the different JPA implementations can differ, however, when it comes to caching, threading, and overall performance. EclipseLink provides a high-performance JPA implementation with many advanced features.

Coherence provides a default entity-based cache store implementation, JpaCacheStore, and a corresponding cache loader, JpaCacheLoader. You can find additional information in the Javadoc for these classes.

To configure a Coherence JpaCacheStore:

1. **Map the Persistent Classes**

2. **Configure JPA**

3. **Configure a Coherence Cache for JPA**

4. **Configure the Persistence Unit**

2.2.2.2.1 Map the Persistent Classes Map the entity classes to the database. This will allow you to load and store objects through the JPA cache store. JPA mappings are standard, and can be specified in the same way for all JPA providers.

You can map entities either by annotating the entity classes or by adding an orm.xml or other XML mapping file. See the JPA provider documentation for more information about how to map JPA entities.

2.2.2.2.2 Configure JPA Edit the persistence.xml file to create the JPA configuration. This file contains the properties that dictate run-time operation.

Set the transaction type to RESOURCE_LOCAL and provide the required JDBC properties for your JPA provider (such as driver, url, user, and password) with the appropriate values for connecting and logging into your database. List the classes that are mapped using JPA annotations in <class> elements. Example 2–2 illustrates a sample persistence.xml file with the typical properties that you can set.
**Example 2–2 Sample persistence.xml File for JPA**

```xml
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="EmpUnit" transaction-type="RESOURCE_LOCAL">
    <provider>
      org.eclipse.persistence.jpa.PersistenceProvider
    </provider>
    <class>com.oracle.coherence.handson.Employee</class>
    <properties>
      <property name="eclipselink.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
      <property name="eclipselink.jdbc.url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
      <property name="eclipselink.jdbc.user" value="scott"/>
      <property name="eclipselink.jdbc.password" value="tiger"/>
    </properties>
  </persistence-unit>
</persistence>
```

**2.2.2.2 Configure a Coherence Cache for JPA**

Create a `coherence-cache-config.xml` file to override the default Coherence settings and define the `JpaCacheStore` caching scheme. The caching scheme should include a `<cachestore-scheme>` element that lists the `JpaCacheStore` class and includes the following parameters.

- **The entity name of the entity being stored.** Unless it is explicitly overridden in JPA, this is the unqualified name of the entity class. **Example 2–3** uses the built-in Coherence macro `{cache-name}` that translates to the name of the cache that is constructing and using the cache store. This works because a separate cache must be used for each type of persistent entity and Coherence ensures that the name of each cache is set to the name of the entity that is being stored in it.

- **The fully qualified name of the entity class.** If the classes are all in the same package and use the default JPA entity names, then you can again use the `{cache-name}` macro for the part that is variable across the different entity types. In this way, the same caching scheme can be used for all of the entities that are cached within the same persistence unit.

- **The persistence unit name.** This should be the same as the name specified in the `persistence.xml` file.

The various named caches are then directed to use the JPA caching scheme. **Example 2–3** is a sample `coherence-cache-config.xml` file that defines a `NamedCache` class named `Employee` that caches instances of the `Employee` class. To define additional entity caches for more classes, add more `<cache-mapping>` elements to the file.

**Example 2–3 Assigning Named Caches to a JPA Caching Scheme**

```xml
<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <!-- Set the name of the cache to be the entity name. -->
      <cache-name>Employee</cache-name>
      <!-- Configure this cache to use the following defined scheme. -->
      <scheme-name>jpa-distributed</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
</cache-config>
```
2.2.2.4 Configure the Persistence Unit  When using the JpaCacheStore class, configure the persistence unit to ensure that no changes are made to entities when they are inserted or updated. Any changes made to entities by the JPA provider are not reflected in the Coherence cache. This means that the entity in the cache will not match the database contents. In particular, entities should not use ID generation, for example, @GeneratedValue, to obtain an ID. IDs should be assigned in application code before an object is put into Coherence. The ID is typically the key under which the entity is stored in Coherence.

Optimistic locking (for example, @Version) should not be used because it might lead to the failure of a database transaction commit transaction. See Caching Data Sources and Sample CacheStore in Oracle Fusion Middleware Developing Applications with Oracle Coherence for more information about how a cache store works, and how to set up your database schema.

When using either the JpaCacheStore or JpaCacheLoader class, L2 ("shared") caching should be disabled in your persistence unit. See the documentation for your provider. In Oracle TopLink, this can be specified on an individual entity with
@Cache(shared=false) or as the default in the persistence.xml file with the following property:

<property name="eclipselink.cache.shared.default" value="false"/>

When using EclipseLink with TopLink Grid, the TopLink Grid implementations will automatically disable L2 caching, optimistic lock checking, and versioning. Essentially, TopLink Grid implementations understand the cache store context in which the persistence unit is being deployed and adjust the configuration accordingly.
This chapter provides more detailed information on how to configure applications running under Coherence*Web so that they can share Coherence cache and session information.

You can find more information on Coherence*Web and how to enable it for applications running on a variety of application servers in Oracle Fusion Middleware Administering HTTP Session Management with Oracle Coherence*Web.

### 3.1 Merging Coherence Cache and Session Information

In Coherence, the cache configuration deployment descriptor provides detailed information about the various caches that can be used by applications within a cluster. Coherence provides a sample cache configuration deployment descriptor, named `coherence-cache-config.xml`, in the root of the `coherence.jar` library.

In Coherence*Web, the session cache configuration deployment descriptor provides detailed information about the caches, services, and attributes used by HTTP session management. Coherence*Web provides a sample session cache configuration deployment descriptor, named `default-session-cache-config.xml`, in the `coherence-web.jar` library. You can use this file as the basis for any custom session cache configuration file you may need to write.

At run time, Coherence uses the first `coherence-cache-config.xml` file that is found in the classpath, and it must precede the `coherence.jar` library; otherwise, the sample `coherence-cache-config.xml` file in the `coherence.jar` file is used.

In the case of Coherence*Web, it first looks for a custom session cache configuration XML file in the classloader that was used to start Coherence*Web. If no custom session cache configuration XML resource is found, then it will use the `default-session-cache-config.xml` file packaged in `coherence-web.jar`.

If your Coherence applications are using Coherence*Web for HTTP session management, the start-up script for the application server and the Coherence cache servers must reference the session cache configuration file—not the cache configuration file. In this case, you must complete these steps:

1. Extract the session cache configuration file from the `coherence-web.jar` library.
2. Merge the cache information from the Coherence cache configuration file into the session cache configuration file.
Note that in the cache scheme mappings in this file, you cannot use wildcards to specify cache names. You must provide, at least, a common prefix for application cache names.

3. Ensure that modified session cache configuration file is used by the Coherence members in the cluster.

The cache and session configuration must be consistent across WebLogic Servers and Coherence cache servers.
This chapter describes how you can integrate Oracle Coherence with Hibernate, an object-relational mapping tool for Java environments. The functionality in Oracle Coherence and Hibernate can be combined such that Hibernate can act as the Coherence cache store or Coherence can act as the Hibernate L2 cache.

Notes: You can find additional information about integrating Coherence and Hibernate in the Coherence Community projects at the following URL:

https://java.net/projects/cohib

This chapter contains the following sections:

- API for HibernateCacheStore and HibernateCacheLoader
- Using Hibernate as a Cache Store for Coherence
- Using Coherence as the Hibernate L2 Cache Provider

4.1 API for HibernateCacheStore and HibernateCacheLoader

Coherence includes a default entity-based cache store implementation, HibernateCacheStore, and a corresponding cache loader implementation, HibernateCacheLoader, in the com.tangosol.coherence.hibernate package.

Table 4–1 describes the different constructors for the HibernateCacheStore and HibernateCacheLoader classes. For more detailed technical information, see the Javadoc for these classes.
4.2 Using Hibernate as a Cache Store for Coherence

Hibernate can also be used as a cache store implementation for Coherence. Applications that use this approach typically have the following characteristics:

- Use Coherence APIs for data access/management
- Have simpler object models appropriate for the out-of-the-box Hibernate CacheStore implementation (note that applications with more complex object models can take advantage of custom CacheStore implementations)
- Have simpler transactional requirements
- Require high performance through the use of Coherence APIs, such as write-behind and aggregations.

### 4.2.1 Configuration Requirements

Hibernate entities accessed by using the HibernateCacheStore module must use the assigned ID generator and also have a defined ID property.

Disable the hibernate.hbm2ddl.auto property in the hibernate.cfg.xml file used by the HibernateCacheStore module to avoid excessive schema updates and possible lockups.

### 4.2.2 Configuring a Hibernate Cache Store Constructor

The following examples illustrate how to configure a simple HibernateCacheStore constructor, which accepts only an entity name. This configures Hibernate by using the default configuration path, which looks for a hibernate.cfg.xml file in the class path. You can also include a resource name or file specification for the
Using Hibernate as a Cache Store for Coherence

Integrating Hibernate and Coherence

Example 4–1 illustrates a simple coherence-cache-config.xml file used to define a NamedCache cache object named TableA that caches instances of a Hibernate entity (com.company.TableA). To define more entity caches, add additional <cache-mapping> elements.

Example 4–1  Coherence Cache Configuration File for Hibernate

```xml
<?xml version="1.0"?>
<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
  xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-config/coherence-cache-config.xsd">
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>TableA</cache-name>
      <scheme-name>distributed-hibernate</scheme-name>
      <init-params>
        <init-param>
          <param-name>entityname</param-name>
          <param-value>com.company.TableA</param-value>
        </init-param>
      </init-params>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <distributed-scheme>
      <scheme-name>distributed-hibernate</scheme-name>
      <backing-map-scheme>
        <read-write-backing-map-scheme>
          <internal-cache-scheme>
            <local-scheme/>
          </internal-cache-scheme>
        </read-write-backing-map-scheme>
        <cachestore-scheme>
          <class-scheme>
            <class-name>com.tangosol.coherence.hibernate.HibernateCacheStore</class-name>
            <init-params>
              <init-param>
                <param-name>entityname</param-name>
                <param-value>com.company.TableA</param-value>
              </init-param>
            </init-params>
          </class-scheme>
        </cachestore-scheme>
      </backing-map-scheme>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
```

Example 4–2 illustrates that you can also use the predefined {cache-name} macro to eliminate the need for the <init-params> portion of the cache mapping.

Example 4–2 illustrates that you can also use the predefined {cache-name} macro to eliminate the need for the <init-params> portion of the cache mapping.
Example 4–2  Coherence Cache Configuration File that Uses (cache-name) Macro

<?xml version="1.0"?>

<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
    xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-config
    coherence-cache-config.xsd">
    <caching-scheme-mapping>
        <cache-mapping>
            <cache-name>TableA</cache-name>
            <scheme-name>distributed-hibernate</scheme-name>
        </cache-mapping>
    </caching-scheme-mapping>

    <caching-schemes>
        <distributed-scheme>
            <scheme-name>distributed-hibernate</scheme-name>
            <backing-map-scheme>
                <read-write-backing-map-scheme>
                    <internal-cache-scheme>
                        <local-scheme/></local-scheme>
                    </internal-cache-scheme>
                    <cachestore-scheme>
                        <class-scheme>
                            <class-name>
                                com.tangosol.coherence.hibernate.HibernateCacheStore
                            </class-name>
                            <init-params>
                                <init-param>
                                    <param-type>java.lang.String</param-type>
                                    <param-value>com.company.{cache-name}</param-value>
                                </init-param>
                            </init-params>
                        </class-scheme>
                    </cachestore-scheme>
                </read-write-backing-map-scheme>
            </backing-map-scheme>
        </distributed-scheme>
    </caching-schemes>
</cache-config>

Example 4–3 illustrates that, if naming conventions allow, the mapping can be completely generalized to enable a cache mapping for any qualified class name (entity name).

Example 4–3  Sample coherence-cache-config.xml File with Generalized Mappings

<?xml version="1.0"?>

<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
    xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-config
    coherence-cache-config.xsd">
    <caching-scheme-mapping>
        <cache-mapping>
            <cache-name>com.company.*</cache-name>
            <scheme-name>distributed-hibernate</scheme-name>
        </cache-mapping>
    </caching-scheme-mapping>
</cache-config>
4.2.3 Creating a Hibernate Cache Store

While the provided HibernateCacheStore module provides a solution for most entity-based caches, there may be cases where an application-specific CacheStore module is necessary. For example, for providing parameterized queries, or including or post-processing query results.

4.2.3.1 Reentrant Calls

In a cache store-backed cache implementation, when the application thread accesses cached data, the cache operations may trigger a call to the associated CacheStore implementation by using the managing CacheService API. The CacheStore must not call back into the CacheService API. This implies, indirectly, that Hibernate should not attempt to access cache data. Therefore, all methods in the CacheLoader or CacheStore implementation should be careful to call Session.setCacheMode(CacheMode.IGNORE) method to disable cache access. Alternatively, the Hibernate configuration can be cloned (either programatically or by using the hibernate.cfg.xml file), with CacheStore implementations using the version with the cache disabled.

4.2.4 Extending the Hibernate Cache Store Functionality

In some cases, you may want to extend the Hibernate cache store with application-specific functionality. The most obvious reason for this is to take advantage of a preexisting, programatically configured SessionFactory instance.
4.2.5 JDBC Isolation Level

In cases where all access to a database is through Coherence, cache store modules naturally enforce ANSI-style repeatable read isolation as read operations, and write operations are executed serially on a per-key basis (by using the Partitioned Cache Service). Increasing database isolation above the repeatable read level does not yield increased isolation because cache store operations might span multiple partitioned cache nodes (and thus multiple database transactions). Using database isolation levels below the repeatable read level does not result in unexpected anomalies, and might reduce processing load on the database server.

4.2.6 Fault-Tolerance for Hibernate Cache Store Operations

For single-cache-entry updates, cache store operations are fully fault-tolerant in that the cache and database are guaranteed to be consistent during any server failure (including failures during partial updates). While the mechanisms for fault-tolerance vary, this is true for both write-through and write-behind caches.

Coherence does not support two-phase cache store operations across multiple cache store instances. In other words, if two cache entries are updated, triggering calls to cache store modules sitting on separate servers, it is possible for one database update to succeed and for the other to fail. In this case, you might want to use a cache-aside architecture (updating the cache and database as two separate components of a single transaction) with the application server transaction manager. In many cases, it is possible to design the database schema to prevent logical commit failures (but obviously not server failures). Write-behind caching avoids this issue because put operations are not affected by database behavior (and the underlying issues have been addressed earlier in the design process).

4.2.7 Using Fully Cached Data Sets

There are two scenarios where using fully cached data sets would be advantageous. One is when you are performing distributed queries on the cache; the other is when you want to provide continued application processing despite a database failure.

4.2.7.1 Distributed Queries

Distributed queries offer the potential for lower latency, higher throughput, and less database server load, as opposed to executing queries on the database server. For set-oriented queries, the data set must be entirely cached to produce correct query results. More precisely, for a query issued against the cache to produce correct results, the query must not depend on any uncached data.

Distributed queries enable you to create hybrid caches. For example, it is possible to combine two uses of NamedCache: a fully cached size-limited data set for querying (for example, the data for the most recent week), and a partially cached historical data set used for singleton read operations. This approach avoids data duplication and minimizes memory usage.

While fully cached data sets are usually bulk-loaded during application startup (or on a periodic basis), cache store integration can be used to ensure that both cache and database are kept fully synchronized.

4.2.7.2 Detached Processing

Another reason for using fully cached data sets is to provide the ability to continue application processing even if the underlying database fails. Using write-behind caching extends this mode of operation to support full read-write applications. With
write-behind, the cache becomes (in effect) the temporary system of record. Should the
database fail, updates are queued in Coherence until the connection is restored. At this
point, all cache changes are sent to the database.

4.3 Using Coherence as the Hibernate L2 Cache Provider

Using Coherence as a Hibernate L2 cache provider enables multiple JVMs running the
same Hibernate application to share an L2 cache. The use of Coherence caching in this
case is controlled by Hibernate. You should have a good understanding of Hibernate
L2 caching to successfully use this provider. For more information on Hibernate as an
L2 cache, see ‘Improving Performance’ chapter of the Hibernate Reference Manual:
http://www.hibernate.org/docs.html

This may be a good fit for applications that have these characteristics:

- Use Hibernate APIs for data access/management.
- Have large/complex object models.
- Have complicated transactional requirements.
- Have a large cluster of application servers running Hibernate that access the same
database

Hibernate supports three primary forms of caching:

- Session cache
- L2 cache
- Query cache

The session cache is responsible for caching records within a Hibernate Session. A
Hibernate Session is a transaction-level cache of persisted data, potentially spanning
multiple database transactions, and typically scoped on a per-thread basis. As a
nonclustered cache (by definition), the session cache is managed entirely by Hibernate.

The L2 and query caches span multiple transactions, and support the use of Coherence
as a cache provider. The L2 cache is responsible for caching records across multiple
sessions (for primary key lookups). The query cache caches the result sets generated
by Hibernate queries. Hibernate manages data in an internal representation in the L2
and query caches, meaning that these caches are usable only by Hibernate. For more
information, see the Hibernate Reference Documentation (shipped with Hibernate),
specifically the section on the Second Level Cache.

4.3.1 Configuring Coherence as the Hibernate L2 Cache

To use the Coherence caching provider for Hibernate, specify the Coherence provider
class in the hibernate.cache.provider_class property. Typically, this is configured
in the default Hibernate configuration file, hibernate.cfg.xml. Example 4–4
illustrates the property element calling CoherenceCacheProvider
(com.tangosol.coherence.hibernate.CacheProvider) as the value of the
hibernate.cache.provider_class property.

Example 4–4  Specifying a Coherence Provider Class

<property name='hibernate.cache.provider_class'>com.tangosol.coherence.hibernate.CoherenceCacheProvider</property>

The coherence-hibernate.jar file (found in the lib/ subdirectory) must be added to
the application class path.
Hibernate provides the configuration property `hibernate.cache.use_minimal_puts`, which optimizes cache access for clustered caches by increasing cache read operations and minimizing cache update operations. The Coherence caching provider enables this by default. Setting this property to false might increase overhead for cache management and also increase the number of transaction rollbacks.

The Coherence caching provider includes a setting for how long a lock acquisition should be attempted before timing out. Use the Java property `tangosol.coherence.hibernate.lockattemptmillis` to specify the value. The default is one minute.

4.3.2 Specifying a Coherence Cache Topology

By default, the Coherence caching provider uses a custom cache configuration located in the `coherence-hibernate.jar` file named `config/hibernate-cache-config.xml`. This configuration file defines cache mappings for Hibernate L2 caches. If desired, you can specify an alternative cache configuration resource for Hibernate L2 caches by using the `tangosol.coherence.hibernate.cacheconfig` Java property. You can configure this property to point to the application's main `coherence-cache-config.xml` file if the mappings are properly configured. It can be beneficial to use dedicated cache service(s) to manage Hibernate-specific caches to ensure that any cache store modules do not cause reentrant calls back into Coherence-managed Hibernate L2 caches.

In the scheme mapping section of the Coherence cache configuration file, the `hibernate.cache.region_prefix` property can specify a cache topology. For example, if the cache configuration file includes a wildcard mapping for `near-*`, and the Hibernate region prefix property is set to `near-`, then all Hibernate caches are named using the `near-` prefix, and use the cache scheme mapping specified for the `near-*` cache name pattern.

It is possible to specify a cache topology per entity by creating a cache mapping based on the combined prefix and qualified entity name (for example, `near-com.company.EntityName`); or equivalently, by providing an empty prefix and specifying a cache mapping for each qualified entity name.

L2 caches should be size-limited to avoid excessive memory usage. Query caches in particular must be size-limited because the Hibernate API does not provide any means of controlling the query cache other than a complete eviction.

4.3.3 Choosing a Cache Concurrency Strategy

Hibernate generally emphasizes the use of optimistic concurrency for both cache and database. With optimistic concurrency in particular, transaction processing depends on having accurate data available to the application at the beginning of the transaction. If the data is inaccurate, then commit processing detects that the transaction was dependent on incorrect data, and the transaction is not committed. While most optimistic transactions must adjust to changes to underlying data by other processes, the use of caching adds the possibility of the cache itself being stale. Hibernate provides several cache concurrency strategies to control updates to the L2 cache. While this is less of an issue for Coherence due to support for clusterwide coherent caches, the appropriate selection of cache concurrency strategy aids application efficiency.

Note that cache configuration strategies can be specified at the table level. Generally, the strategy should be specified in the mapping file for the class.
For mixed read-write activity, the read-write strategy is recommended. The transactional strategy is implemented similarly to the nonstrict-read-write strategy, and relies on the optimistic concurrency features of Hibernate. A nonstrict-read-write strategy is appropriate when the application only occasionally updates data, and does not require strict transaction isolation. Note that the nonstrict-read-write may deliver better performance if its impact on optimistic concurrency is acceptable.

For read-only caching, use the nonstrict-read-write strategy if the underlying database data might change, but slightly stale data is acceptable. If the underlying database data never changes, use the read-only strategy.

### 4.3.4 Query Cache

To cache query results, set the `hibernate.cache.use_query_cache` property to `true`. Then, whenever issuing a query that must cache its results, use the `Query.setCacheable(true)` method. As `org.hibernate.cache.QueryKey` instances in Hibernate might not be binary-comparable (due to nondeterministic serialization of unordered data members), use a size-limited local or replicated cache to store query results (this forces the use of the `hashcode()` or `equals()` methods to compare keys).

The default query cache name is `org.hibernate.cache.StandardQueryCache` (unless a default region prefix is provided; in this case `[prefix]` is prefixed to the cache name). Use the cache configuration file to map this cache name to a local or replicated topology, or to explicitly provide an appropriately-mapped region name when querying.

### 4.3.5 Fault-Tolerance for Hibernate L2 Caches

The Hibernate L2 cache protocol supports full fault-tolerance during client or server failure. With the read-write cache concurrency strategy, Hibernate locks items out of the cache at the start of an update transaction, meaning that client-side failures simply result in uncached entities and an uncommitted transaction. Server-side failures are handled transparently by Coherence (dependent on the specified data backup count).

### 4.3.6 Deployment

When used with application servers that do not have a unified class loader, the Coherence caching provider must be deployed as part of the application so that it can use the application-specific class loader (required to serialize and deserialize objects).
This chapter describes where you can find information on integrating Oracle Coherence with Spring, a platform for building and running Java-based enterprise applications. You can find information on how to configure the Oracle Coherence cache to consume objects provided by the Spring platform in Coherence Community projects. Coherence Community projects provide example implementations for commonly used design patterns based on Oracle Coherence. See the following URL:

https://java.net/projects/cohspr/
Enabling ECID in Coherence Logs

This chapter describes how Oracle Coherence can use the Execution Context ID (ECID). This globally unique ID can be attached to requests between Oracle components. The ECID allows you to track log messages pertaining to the same request when multiple requests are processed in parallel.

Coherence logs will include ECID only if the client already has an activated ECID prior to calling Coherence operations. The ECID may be passed from another component or obtained in the client code. To activate the context, use the get and activate methods on the oracle.dms.context.ExecutionContext interface in the Coherence client code. The ECID will be attached to the executing thread. Use the deactivate method to release the context, for example:

```java
Example 6–1 Using a DMS Context in Coherence Client Code
...
// Get the context associated with this thread
ExecutionContext ctx = ExecutionContext.get();
ctx.activate();
...
set additional execution context values (optional)
perform some cache operations
...
// Release the context
ctx.deactivate();
...
```

ECID logging will occur only on the node where the client is running. If a client request is processed on some other node and an exception is thrown by Coherence, then the remote error will be returned to the originating node and it will be logged on the Coherence client. The log message will contain the ECID of the request. Messages logged on the remote node will not contain the ECID.

For more information on how to include the ECID in a Coherence log message, see “Changing the Log Message Format” in the Oracle Fusion Middleware Developing Applications with Oracle Coherence.
Integrating with Oracle Coherence GoldenGate HotCache

This chapter describes how to use Oracle Coherence GoldenGate HotCache (HotCache) with applications using Coherence caches. HotCache allows changes to the database to be propagated to objects in the Coherence cache.

A detailed description Oracle GoldenGate is beyond the scope of this documentation. For more information, see Oracle GoldenGate for Oracle Installation and Setup Guide to install GoldenGate on Oracle databases and Oracle GoldenGate for Java Administrator’s Guide.

Note: To use HotCache, you must have licenses for Oracle GoldenGate and Coherence Grid Edition. HotCache can be used with Oracle GoldenGate 11gR1 and 11gR2 releases.

This chapter contains the following sections:

- Overview
- How Does HotCache Work?
- Prerequisites
- Configuring GoldenGate
- Configuring HotCache
- Configuring the GoldenGate Java Client
- Using Portable Object Format with HotCache
- Enabling Wrapper Classes for TopLink Grid Applications

7.1 Overview

Third-party updates to the database can cause Coherence applications to work with data which could be stale and out-of-date. HotCache solves this problem by monitoring the database and pushing any changes into the Coherence cache. HotCache employs an efficient push model which processes only stale data. Low latency is assured because the data is pushed when the change occurs in the database.

HotCache can be added to any Coherence application. Standard JPA is used to capture the mappings from database data to Java objects. The configuration can be captured in XML exclusively or in XML with annotations.
The following scenario describes how HotCache could be used to work with the database and with applications that use Coherence caches. Figure 7–1 illustrates the scenario.

1. Start GoldenGate—see "Starting the Application" in Oracle GoldenGate for Java Administrator’s Guide for details. GoldenGate monitors the transaction log for changes of interest. These changes will be placed into a "trail file".

2. Start the Coherence cache server and warm the cache, if required.

3. Start HotCache so that it can propagate changes in the trail file into the cache. If changes occur during cache warming, then they will be applied to the cache once HotCache is started so no changes are lost.

4. Start an application client. As part of its operation, assume that the application performs repeated queries on the cache.

5. A third-party application performs a direct database update.

6. GoldenGate detects the database change which is then propagated to the Coherence cache by HotCache.

7. The application client detects the change in cache.

**Figure 7–1  How HotCache Propagates Database Changes to the Cache**

---

### 7.2 How Does HotCache Work?

HotCache processes database change events delivered by GoldenGate and maps those changes onto the affected objects in the Coherence cache. It is able to do this through the use of Java Persistence API (JPA) mapping metadata. JPA is the Java standard for object-relational mapping in Java and it defines a set of annotations (and corresponding XML) that describe how Java objects are mapped to relational tables.

As Example 7–1 illustrates, instances of an Employee class could be mapped to rows in an EMPLOYEE table with the following annotations.

**Example 7–1  Mapping Instances of Employee Class to Rows with Java Code**

```java
@Entity
@Table(name="EMPLOYEE")
public class Employee {
    @Id
    @Column(name="ID")
```
The @Entity annotation marks the Employee class as being persistent and the @Id annotation identifies the id field as containing its primary key. In the case of Coherence cached objects, the @Id field must also contain the value of the key under which the object is cached. The @Table and @Column annotations associate the class with a named table and a field with a named column, respectively.

For simplification, JPA assumes a number of default mappings such as table name=class name and column name=field name so many mappings need only be specified when the defaults are not correct. In Example 7–1, both the table and field names match the Java names so the @Table and @Column can be removed to make the code more compact, as illustrated in Example 7–2.

**Example 7–2  Simplified Java Code for Mapping Instances of Employee Class to Rows**

```java
@Entity
public class Employee {
    @Id
    private int id;
    private String firstName;
    ...
}
```

Note that the Java code in the previous examples can also be expressed as XML. Example 7–3 illustrates the XML equivalent of the Java code in Example 7–1.

**Example 7–3  Mapping Instances of Employee Class to Rows with XML**

```xml
<entity class="Employee">
    <table name="EMPLOYEE"/>
    <attributes>
        <id name="id">
            <column name="ID"/>
        </id>
        <basic name="firstName">
            <column name="FIRSTNAME"/>
        </basic>
    .....
    </attributes>
</entity>
```

Similarly, Example 7–4 illustrates the XML equivalent for the simplified Java code in Example 7–2.

**Example 7–4  Simplified XML for Mapping Instances of Employee Class to Rows**

```xml
<entity class="Employee">
    <attributes>
        <id name="id"/>
        <basic name="firstName"/>
    .....
    </attributes>
</entity>
```
7.2.1 How the GoldenGate Java Adapter uses JPA Mapping Metadata

JPA mapping metadata provides mappings from object to relational; however, it also provides the inverse relational to object mappings which HotCache can use. Given the Employee example, consider an update to the FIRSTNAME column of a row in the EMPLOYEE table. Figure 7–2 illustrates the EMPLOYEE table before the update, where the first name John is associated with employee ID 1, and the EMPLOYEE table after the update where first name Bob is associated with employee ID 1.

Figure 7–2 EMPLOYEE Table Before and After an Update

Before:

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRSTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRSTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
</tr>
</tbody>
</table>

With GoldenGate monitoring changes to the EMPLOYEE table and to the HotCache configured on the appropriate trail file, the adapter will process an event indicating the FIRSTNAME column of the EMPLOYEE row with primary key 1 has been changed to Bob. The adapter will use the JPA mapping metadata to first identify the class associated with the EMPLOYEE table, Employee, and then determine the column associated with an Employee's ID field, ID. With this information, the adapter can extract the ID column value from the change event and update the firstName field (associated with the FIRSTNAME column) of the Employee cached under the ID column value.

7.2.2 Supported Database Operations

Database INSERT, UPDATE, and DELETE operations are supported by the GoldenGate Java Adapter. INSERT operations into a mapped table will result in the addition of a new instance of the associated class populated with the data from the newly inserted row. Changes applied through an UPDATE operation will be propagated to the corresponding cached object. If the cache does not contain an object corresponding to the updated row, the cache is unchanged. A DELETE operation will result in the removal of the corresponding object from the cache, if one exists.

7.3 Prerequisites

The instructions in the following sections assume that you have set up your database to work with GoldenGate. This includes the following tasks:

- creating a database and tables
Configuring GoldenGate

- granting user permissions
- enabling logging
- provisioning the tables with data

Example 7–5 illustrates a list of sample commands that creates a user named csdemo and grants user permissions to the database.

Note the ALTER DATABASE ADD SUPPLEMENTAL LOG DATA command. When supplemental logging is enabled, all columns are specified for extra logging. At the very least, minimal database-level supplemental logging must be enabled for any change data capture source database.

**Example 7–5  Sample Commands to Create a User, Grant Permissions, and Enable Logging**

CREATE USER csdemo IDENTIFIED BY csdemo;
GRANT DBA TO csdemo;
grant alter session to csdemo;
grant create session to csdemo;
grant flashback any table to csdemo;
grant select any dictionary to csdemo;
grant select any table to csdemo;
grant select any transaction to csdemo;
grant unlimited tablespace to csdemo;
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;

The instructions in the following sections also assume that you have installed Oracle GoldenGate and started the manager. This includes the following tasks:

- downloading and installing Oracle GoldenGate
- running ggsci to create the GoldenGate subdirectories
- creating a manager parameter (mgr.prm) file, specifying the listener port
- adding JVM libraries to the libraries path
- starting the manager

A detailed description of these tasks is beyond the scope of this documentation. For more information, see *Oracle GoldenGate for Oracle Installation and Setup Guide* to install GoldenGate on Oracle databases and *Oracle GoldenGate for Java Administrator’s Guide*.

### 7.4 Configuring GoldenGate

Updating the cache from a GoldenGate trail file requires configuring GoldenGate and HotCache. You then enable HotCache by configuring the GoldenGate Java Adapter.

- **Monitor Table Changes**
- **Filter Changes Made by the Current User**

#### 7.4.1 Monitor Table Changes

Indicate the table that you want to monitor for changes by using the ADD TRANDATA command. The ADD TRANDATA command can be used on the command line or as part of a ggsci script. For example, to monitor changes to tables in the csdemo schema, use the following command:

ADD TRANDATA csdemo.*
Example 7–6 illustrates a sample ggsci script named cs-cap.ggsci.

- The script starts the manager and logs into the database. It stops and deletes any running extract named cs-cap.
- The ADD TRANDATA command instructs the extract that tables named csdemo* should be monitored for changes.
- The SHELL command deletes all trail files in the dirdat directory to ensure that if the extract is being recreated, there will be no old trail files. Note that the rm -f command is platform-specific. An extract named cs-cap is created using parameters from the dirprm/cs-cap.prm file. A trail is added at dirdat/cs from the extract cs-cap file.
- The start command starts the cs-cap.ggsci script.
- The ADD EXTRACT command automatically uses the cs-cap.prm file as the source of parameters, so a PARAMS dirprm/cs-cap.prm statement is not necessary.

**Example 7–6  Sample GoldenGate Java Adapter ggsci Script to Monitor Table Changes**

```plaintext
start mgr
DBLOGIN USERID csdemo, PASSWORD csdemo
STOP EXTRACT cs-cap
DELETE EXTRACT cs-cap
ADD TRANDATA csdemo.*
ADD EXTRACT cs-cap, tranlog, begin now
SHELL rm -f dirdat/cs*
ADD EXTTRAIL dirdat/cs, EXTRACT cs-cap
start cs-cap
```

### 7.4.2 Filter Changes Made by the Current User

Configure GoldenGate to ignore changes made by the user that the Coherence CacheStores are logged in as. This avoids GoldenGate processing any changes made to the database that are already in the cache.

The **TranLogOptions excludeUSER** command can be used on the command line or in a ggsci script. For example, the following command instructs GoldenGate extract process to ignore changes to the database tables made by the Coherence CacheStore user logged in as csdemo.

```plaintext
TranLogOptions excludeUser csdemo
```

**Example 7–7 illustrates a sample extract .prm file named cs-cap.prm.** The user that the Coherence CacheStore is logged in as is csdemo. The **recoveryOptions OverwriteMode** line indicates that the extract overwrites the existing transaction data in the trail after the last write-checkpoint position, instead of appending the new data. The **EXTRAIL** parameter identifies the trail as dirdat/cs. The **BR BROFF** parameter controls the Bounded Recovery (BR) feature. The **BROFF** value turns off Bounded Recovery for the run and for recovery. The **GETUPDATEBEFORES** parameter indicates that the before images of updated columns are included in the records that are processed by Oracle GoldenGate. The **TABLE** parameter identifies csdemo.* as the tables that should be monitored for changes. The **TranLogOptions excludeUSER** parameter indicates that GoldenGate should ignore changes to the tables made by the Coherence CacheStore user logged in as csdemo.

**Example 7–7  Sample Extract .prm File for the GoldenGate Java Adapter**

```plaintext
EXTRACT cs-cap
USERID csdemo, PASSWORD csdemo
```
7.5 Configuring HotCache

HotCache is configured with system properties, EclipseLink JPA mapping metadata (as described in "How Does HotCache Work?" on page 7-2), and a JPA persistence.xml file. The connection from HotCache to the Coherence cluster is made by using Coherence*Extend (TCP).

The following sections describe the properties needed to configure HotCache and provides details about connecting with Coherence*Extend.

- Create a Properties File with GoldenGate for Java Properties
- Add Java Boot Options to the Properties File
- Provide Coherence*Extend Connection Information

7.5.1 Create a Properties File with GoldenGate for Java Properties

Create a text file with the filename extension .properties. In the file, enter the configuration for HotCache. A minimal configuration should contain the list of event handlers, the fully-qualified Java class of the event handler, whether the user-exit checkpoint file is being used, and the name of the Java writer.

Example 7–8 illustrates a .properties file that contains the minimal configuration for a HotCache project. The following properties are used in the file:

- `gg.handlerlist=cgga`
  The `gg.handlerlist` property specifies a comma-separated list of active handlers. This example defines the logical name `cgga` as database change event handler. The name of a handler can be defined by the user, but it must match the name used in the `gg.handler.{name}.type` property in the following bullet.

- `gg.handler.cgga.type=oracle.toplink.goldengate.CoherenceAdapter`
  The `gg.handler.{name}.type` property defines handler for HotCache. The `{name}` field should be replaced with the name of an event handler listed in the `gg.handlerlist` property. The only handler that can be set for HotCache is `oracle.toplink.goldengate.CoherenceAdapter`.

- `goldengate.userexit.nochkpt=true`
  The `goldengate.userexit.nochkpt` property is used to disable the user-exit checkpoint file. This example defines the user-exit checkpoint file to be disabled.
Configuring HotCache

- `goldengate.userexit.writers=jvm`  
The `goldengate.userexit.writers` property specifies the name of the writer. The value of `goldengate.userexit.writers` must be `jvm` to enable calling out to the GoldenGate Java Adapter.

- `-Dlog4j.configuration=my-log4j.properties`  
The `-Dlog4j.configuration` property specifies a user-defined Log4J properties file, `my-log4j.properties`, in the `dirprm` directory (note that the `dirprm` directory is already on the classpath). This statement is optional, because properties can be loaded from classpath (that is, `log4j.configuration=my-log4j.properties`). For more information on configuring logging properties for HotCache, see "Logging Properties" on page 7-10.

There are many other properties that can be used to control the behavior of the GoldenGate Java Adapter. For more information, see the Oracle GoldenGate for Java Administrator’s Guide.

**Example 7–8 .properties File for a HotCache Project**

```
# ====================================================================
# List of active event handlers. Handlers not in the list are ignored.
# ====================================================================
gg.handlerlist=cgga

# ====================================================================
# Coherence cache updater
# ====================================================================
gg.handler.cgga.type=oracle.toplink.goldengate.CoherenceAdapter

# ====================================================================
# Native JNI library properties
# ====================================================================
goldengate.userexit.nochkpt=true

goldengate.userexit.writers=jvm

# ====================================================================
# Java boot options
# ====================================================================
```

The Java boot options are described in the following section.

### 7.5.2 Add Java Boot Options to the Properties File

This section describes the properties that must appear in the Java boot options section of the `.properties` file. These options are defined by using the `jvm.bootoptions` property.
A sample `jvm.bootoptions` listing is illustrated in the **Java boot options** section of Example 7–8.

- **Java Classpath Files**
- **HotCache-related Properties**
- **Coherence-related Properties**
- **Logging Properties**

### 7.5.2.1 Java Classpath Files
The following is a list of directories and JAR files that should be included in the Java classpath. The directories and JAR files are defined with the `java.class.path` property.

- **dirprm** — the GoldenGate `dirprm` directory which contains the extract `.prm` files
- **ggjava.jar** — contains the GoldenGate Java adapter libraries
- **coherence.jar** — contains the Oracle Coherence libraries
- **javax.persistence_2.0.0.0_2-0.jar** — contains the Java persistence libraries
- **eclipselink.jar** — contains the EclipseLink libraries
- **toplink-grid.jar** — contains the HotCache files and the Oracle TopLink libraries
- **domain classes** — the JAR file or directory containing the user classes cached in Coherence that are mapped with JPA for use in HotCache. Also, the Coherence configuration files, `persistence.xml` file, and any `orm.xml` file.

### 7.5.2.2 HotCache-related Properties
The `toplink.goldengate.persistence-unit` property is required as it identifies the persistence unit defined in `persistence.xml` file that HotCache should load. The persistence unit contains information such as the list of participating domain classes, configuration options, and optionally, database connection information.

The `toplink.goldengate.on-error` property is optional. It controls how the adapter responds to errors while processing a change event. This response applies to both expected optimistic lock exceptions and to unexpected exceptions. This property is optional, as its value defaults to "Refresh". Refresh causes the adapter to attempt to read the latest data for a given row from the database and update the corresponding object in the cache. Refresh requires a database connection to be specified in the `persistence.xml` file. This connection will be established during initialization of HotCache. If a connection cannot be made, then an exception is thrown and HotCache will fail to start.

The other on-error strategies do not require a database connection. They are:

- **Ignore** — Log the exception only. The cache may be left with stale data. Depending on application requirements and cache eviction policies this may be acceptable.
- **Evict** — Log a warning and evict the object corresponding to the change database row from the cache
- **Fail** — Throw an exception and exit HotCache

### 7.5.2.3 Coherence-related Properties
Any Coherence property can be passed as a system property in the Java boot options. The `tangosol.coherence.distributed.localstorage` system property with a value of `false` is the only Coherence property that is required to be passed in the Java boot
options. Like all Coherence properties, precede the property name with the -D prefix in the jvm.bootoptions statement, for example:
-Dtangosol.coherence.distributed.localstorage=false

7.5.2.4 Logging Properties
The following logging properties can be defined for HotCache.

The -Dlog4j.configuration=default-log4j.properties property specifies the default Log4J configuration file. Example properties are located in $GOLDEN_GATE_HOME/ggjava/resources/classes/ directory. You can merge these with your existing Log4J configuration.

The Log4J properties file that is bundled with GoldenGate for Java is for demonstration purposes only. The file can be used as-is, or you can merge its contents with the existing Log4J properties.

If the file is used as-is, then it should be copied into the dirprm directory, given a new name, and specified with the -Dlog4j.configuration property. For example, the following line specifies the user-defined my-log4j.properties file in the dirprm directory (note the dirprm directory is already on the classpath):
-Dlog4j.configuration=my-log4j.properties

Using the default properties file in its current location can cause problems during upgrades: your changes will lost when a new distribution is installed.

To allow HotCache to log warnings, add the following line to the property file:
log4j.logger.oracle.toplink.goldengate=WARN, stdout, rolling

To allow HotCache to log errors, add the following line to the property file you use:
-Dlog4j.logger.oracle.toplink.goldengate=DEBUG, stdout, rolling

---

Note: A Coherence Log4J configuration can co-exist with the GoldenGate Log4J configuration. Both can be included in the same file that is configured on the jvm.bootoptions path.

---

7.5.3 Provide Coherence*Extend Connection Information
Oracle recommends that the connection between HotCache and the Coherence cluster be made with Coherence*Extend. For more information on Coherence*Extend, see Oracle Fusion Middleware Developing Remote Clients for Oracle Coherence.

The Coherence configuration files must be in a directory referenced by the jvm.bootoptions=-Djava.class.path= ... entry in the .properties file. For an example, see the jvm.bootoptions entry in Example 7–8.

Example 7–9 illustrates the section of a client cache configuration file that uses Coherence*Extend to connect to the Coherence cluster. In the client cache configuration file, Coherence*Extend is configured in the <remote-cache-scheme> section. By default, the connection port for Coherence*Extend is 9099.

---

Example 7–9  Coherence*Extend Section of a Client Cache Configuration File
<cache-config>
...
  <caching-schemes>
    <remote-cache-scheme>

Example 7–10 illustrates the section of a server cache configuration file that listens for Coherence*Extend connections. In the server cache configuration file, Coherence*Extend is configured in the <proxy-scheme> section. By default, the listener port for Coherence*Extend is 9099.

Example 7–10  Coherence*Extend Section of a Server Cache Configuration File

Example 7–10  Coherence*Extend Section of a Server Cache Configuration File

7.6 Configuring the GoldenGate Java Client

The GoldenGate Java client provides a way to process GoldenGate data change events in Java by configuring an event handler class. The configuration for the GoldenGate Java client allows it to monitor an extract and to pass data change events to HotCache.

This configuration is provided in an extracts .prm file and is described in the following section. The extracts .prm file also contains a reference to the event handler class. This is the same event handler class that is specified in the properties file described in "Create a Properties File with GoldenGate for Java Properties" on page 7-7.
7.6.1 Edit the GoldenGate Java Client Extracts File

This section describes the parameters that can be defined in the extract .prm file for a GoldenGate Java client. The parameters are illustrated in Example 7–11 and constitute a minimal configuration for a HotCache project.

For more information on these parameters and others that can be added to a .prm extracts file, see Oracle GoldenGate for Windows and UNIX Reference Guide.

- **SetEnv (GGS_USEREXIT_CONF = "dirprm/cs-cgga.properties")**
  
  The GGS_USEREXIT_CONF property provides a reference to the .properties file that you created in "Create a Properties File with GoldenGate for Java Properties" on page 7-7. It is assumed that the file is named cs-cgga.properties and is stored in the dirprm folder.

- **GETUPDATEBEFORES**
  
  The GETUPDATEBEFORES property indicates that the before and after values of columns that have changed are written to the trail if the before values are present and can be compared. If before values are not present only after values are written.

- **CUserExit libggjava_ue.so CUSEREXIT PassThru IncludeUpdateBefores**
  
  The CUSEREXIT parameter includes the following:
  
  - The location of the user exit library, which will be libggjava_ue.so for UNIX or ggjava_ue.dll for Windows
  - CUSEREXIT—the callback function name that must be uppercase
  - PASSTHRU—avoids the need for a dummy target trail
  - INCLUDEUPDATEBEFORES—needed for transaction integrity

- **NoTcpSourceTimer**
  
  Use the NOTCPSOURCETIMER parameter to manage the timestamps of replicated operations for reporting purposes within the Oracle GoldenGate environment. NOTCPSOURCETIMER retains the original timestamp value. Use NOTCPSOURCETIMER when using timestamp-based conflict resolution in a bidirectional configuration.

Example 7–11 illustrates a sample .prm file for GoldenGate for Java client.

**Example 7–11 Sample .prm Parameter File for GoldenGate for Java Client**

```
Extract cs-cgga
USERID csdemo, PASSWORD csdemo
SetEnv ( GGS_USEREXIT_CONF = 'dirprm/cs-cgga.properties' )
--- the user-exit library (unix/linux)
CUserExit libggjava_ue.so CUSEREXIT PassThru IncludeUpdateBefores
--- the user-exit library (windows)
--- CUserExit ggjava_ue.dll CUSEREXIT PassThru IncludeUpdateBefores
--- pass all trail data to user-exit (don't ignore/omit/filter data)
GetUpdateBefores
--- TcpSourceTimer (default=on) adjusts timestamps in replicated records for more
--- accurate lag calculation, if time differences between source/target
NoTcpSourceTimer
--- pass all data in trail to user-exit. Can wildcard tables, but not schema name
Table csdemo.*;
```

---

---

---
7.7 Using Portable Object Format with HotCache

This feature was added as part of a patch release. For information about which features were added in a patch, see Release Notes for Oracle Coherence.

Serialization is the process of encoding an object into a binary format. It is a critical component to working with Coherence as data must be moved around the network. Portable Object Format (also known as POF) is a language-agnostic binary format. POF was designed to be very efficient in both space and time and has become a cornerstone element in working with Coherence.

POF serialization can be used with HotCache but requires a small update to the POF configuration file pof-config.xml to allow for TopLink Grid framework classes to be registered.

Classes that participate in POF serialization must be registered in the pof-config.xml file. The POF configuration file has a <user-type-list> element that contains a list of classes that implement PortableObject or have a PofSerializer associated with them. In the case of TopLink Grid, you must register TopLinkPortableObject as the class and TopLinkGridSerializer as the serializer. The <type-id> for each class must be unique, and must match across all cluster instances. For more information on configuring a POF file, see "Registering POF Objects" in Oracle Fusion Middleware Developing Applications with Oracle Coherence.

The <allow-interfaces> element must be set to true to allow you to register a single class for all implementors of the TopLinkPortableObject interface.

Example 7–12 illustrates a sample pof-config.xml file for HotCache. The value integer_value represents a unique integer value greater than 1000.

Example 7–12  Sample POF Configuration File for HotCache

```xml
  <user-type-list>
    <include>coherence-pof-config.xml</include>
    <!-- User types must be above 1000 -->
    ...
    <user-type>
      <type-id><integer_value></type-id>
      <class-name>oracle.eclipselink.coherence.integrated.cache.TopLinkGridPortableObject</class-name>
      <serializer>
        <class-name>oracle.eclipselink.coherence.integrated.cache.TopLinkGridSerializer</class-name>
      </serializer>
    </user-type>
    ...
  </user-type-list>
  <allow-interfaces>true</allow-interfaces>
  ...
</pof-config>
```

7.8 Enabling Wrapper Classes for TopLink Grid Applications

This feature was added as part of a patch release. For information about which features were added in a patch, see Release Notes for Oracle Coherence.
TopLink Grid applications which depend on HotCache to pump changed data to the Coherence cache can use the `toplink.goldengate.enable-toplinkgrid-client` context property set to `true` to generate Java wrapper classes for Coherence cache inserts.

TopLink Grid depends on wrappers to encode relationship information so that eager and lazy JPA relationships can be rebuilt when retrieved from Coherence by TopLink Grid JPA clients. If you are using TopLink Grid with HotCache and the property is not set to `true`, then relationships between objects will be null when retrieved from the Coherence cache.

This context property can be set in the `persistence.xml` file or as a system property in the Java boot options section of the HotCache properties file.
This chapter describes how Oracle Coherence integrates with Oracle WebLogic Portal. Specifically, Coherence includes the following integration points: Coherence*Web for HTTP session state management, a Personalization (P13N) CacheProvider SPI implementation, and a blueprint for efficiently sharing data between Web Services for Remote Portlets (WSRP)-federated portals that use Coherence and the WebLogic Portal Custom Data Transfer mechanism.

This chapter contains the following sections:

- P13N CacheProvider SPI Implementation
- Sharing Data Between WSRP-Federated Portals Using Coherence

### 8.1 P13N CacheProvider SPI Implementation

Internally, WebLogic Portal uses its own caching service to cache portal, personalization, and commerce data as described at the following URL:


WebLogic Portal 8.1.6 and later includes an SPI for the P13N caching service that can be implemented by third-party cache vendors. Coherence includes a P13N cache provider SPI implementation (com.tangosol.coherence.weblogic.PortalCacheProvider) that, when installed into a WebLogic Portal application, transparently manages cached P13N data without requiring code changes. Combining Coherence and WebLogic Portal also gives you more flexibility in your choice of cache topologies.

For example, if you find that your Portal servers are exceeding the 4 GB heap limit (on 32-bit JVMs) or are experiencing slow garbage collection (GC) times, you can use a cache client/server topology to move the serializable P13N state out of your Portal JVMs and into one or more dedicated Coherence cache servers. This reduces your Portal JVM heap size and GC times. Also, you can use the Coherence Management Framework to closely monitor statistics to better tune your P13N cache settings.

Finally, the Coherence cache provider also allows your portlets to use Coherence caching service by using the standard P13N Cache API.

To install the Coherence P13N cache provider:

1. Copy the coherence-wlp.jar and coherence.jar libraries included from the lib directory of the Coherence installation to the APP-INF/lib directory of your WebLogic Portal application.

2. Configure the Coherence P13N cache provider, PortalCacheProvider, as the default provider in the p13n-cache-config.xml file found in each of your...
WebLogic Portal application’s META-INF directory. Specifically, add the following line immediately before the first `<cache>` element:

```xml
<default-provider-id>com.tangosol.coherence.weblogic</default-provider-id>
```

See the Javadoc for the `PortalCacheProvider` class for details on configuring the Coherence `CacheProvider` and Coherence caches used by the provider.

See the following document for a list of caches used by WebLogic Portal:

http://download.oracle.com/docs/cd/E15919_01/wlp.1032/e14227/toc.htm

### 8.2 Sharing Data Between WSRP-Federated Portals Using Coherence

The WSRP protocol was designed to support the federation of portals hosted by arbitrary portal servers and server clusters. Use WSRP to aggregate content and the user interface (UI) from various portlets hosted by other remote portals. By itself, WSRP does not address the challenge of implementing scalable, reliable, and high-performance federated portals that create, access, and manage the life cycle of data shared by distributed portlets. However, WebLogic Portal provides an extension to the WSRP specification that, when coupled with Oracle Coherence, allows WSRP consumers and producers to create, view, modify, and control concurrent access to shared, scoped data in a scalable, reliable, and high-performance manner.

For more information, see the following URL:

http://download.oracle.com/docs/cd/E15919_01/wlp.1032/e14235/toc.htm