Oracle® Coherence

Integration Guide for Oracle TopLink with Coherence Grid

Release 3.7.1

E23131-01

September 2011

ORACLE®

Oracle Coherence Integration Guide for Oracle TopLink with Coherence Grid, Release 3.7.1

E23131-01

Primary Author: Tom Pfaeffle

Contributing Author: Shaun Smith, Gordon Yorke, Rick Sapir

Contents

List of Examples

List of Figures

List of Tables

Preface

Oracle TopLink includes tight integration with Oracle Coherence. This integration, provided through the TopLink Grid feature, blends the standardization and simplicity of application development using the Java Persistence API (JPA) with the scalability and distributed processing power of Oracle Coherence data grid.

This document describes how to:

- Configure TopLink Grid to use the Coherence data grid as the primary data store for entities
- Use Coherence as a distributed shared cache
- Employ Coherence parallel processing to perform Java Persistence Query Language (JPQL) queries on cached entities
- Use the cache store and cache loader interfaces, which have been optimized for EclipseLink JPA, in Coherence applications that run on the grid

Audience

This guide is intended for developers who build applications using JPA and want to use the power of the data grid for improved scalability and performance.

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 and Oracle TopLink, see the following:

- *Oracle Fusion Middleware Developer's Guide for Oracle TopLink*
- *Integration Guide for Oracle Coherence*

- *Getting Started for Oracle Coherence*
- *Developers Guide for Oracle Coherence*
- *Client Guide for Oracle Coherence*
- *Tutorial for Oracle Coherence*
- *User's Guide for Oracle Coherence*Web*
- *Oracle Coherence Management Guide*
- *Oracle Coherence Administrator's Guide*
- *Oracle Coherence Security Guide*

Conventions

The following text conventions are used in this guide:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to Toplink Grid with Oracle Coherence

Oracle TopLink enables you to scale out 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 JP QL 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.

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 JP QL 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 TopLink JPA implementation (EclipseLink) with Oracle Coherence and provides these development approaches:

- 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 the *Integration Guide for Oracle Coherence*.

- 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 Chapter 2, "JPA on the Grid Configurations" 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 Chapter 3, "EclipseLink Native ORM Configurations" for more information.

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.

2

JPA on the Grid Configurations

This chapter contains the following sections:

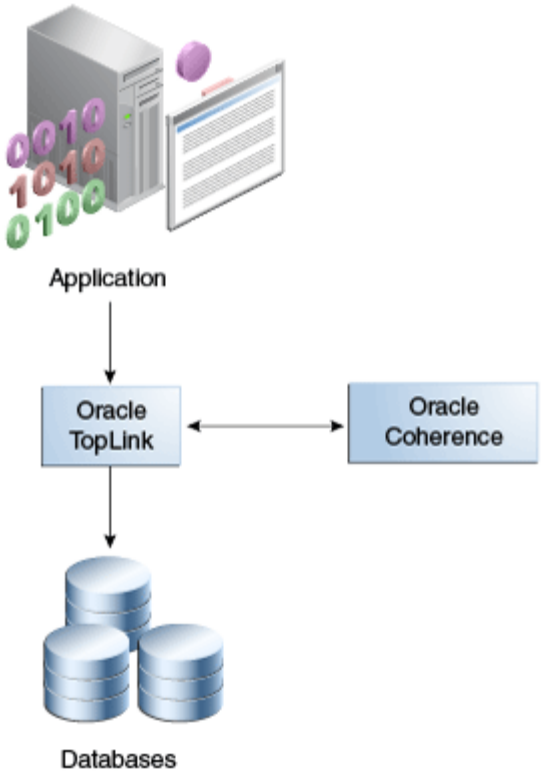
Understanding *JPA on the Grid*

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 2–1 illustrates the relationship between an application, TopLink, Coherence, and the database.

Figure 2–1 JPA on the Grid Approach



JPA on the Grid API

The API used by *JPA on the Grid* configurations are shipped in the `toplink-grid.jar` file. Table 2–1 lists some of the key classes in the `oracle.eclipselink.coherence.integrated` package that are used in *JPA on the Grid* configurations.

Table 2–1 TopLink Grid Classes to Build JPA on the Grid Applications

Class Name	Description
`oracle.eclipseLink.coherence.integrated.EclipseLinkJPACacheLoader`	Provides JPA-aware versions of the Coherence `CacheLoader` interface.
`oracle.eclipseLink.coherence.integrated.EclipseLinkJPACacheStore`	Provides JPA-aware versions of the Coherence `CacheStore` interface.
`oracle.eclipselink.coherence.integrated.config.CoherenceReadCustomizer`	Enables a Coherence read configuration.
`oracle.eclipselink.coherence.integrated.config.CoherenceReadWriteCustomizer`	Enables a Coherence read/write configuration.
`oracle.eclipselink.coherence.integrated.config.GridCacheCustomizer`	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.
`oracle.eclipselink.coherence.integrated.querying.IgnoreDefaultRedirector`	Allows queries to bypass the Coherence cache and be sent directly to the database.

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.

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 2-4.

See "Grid Cache Configuration Examples" on page 2-5 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`

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

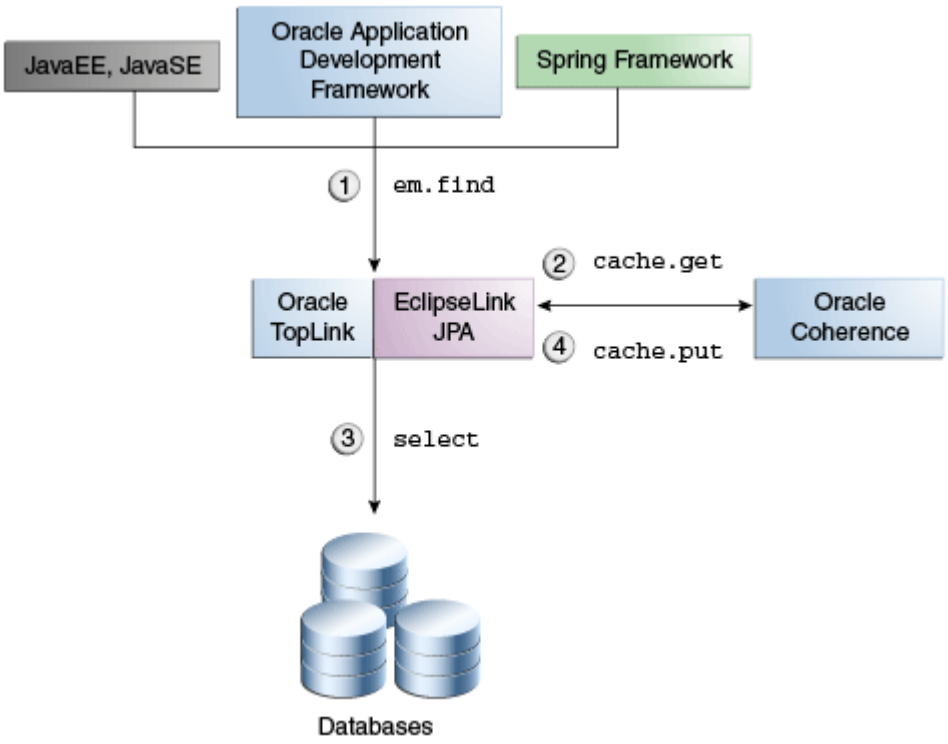
entity is used and a new entity is not built. This approach can greatly improve application performance, especially with a warmed cache, because it reduces the cost of a query by eliminating the cost associated with object building.

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



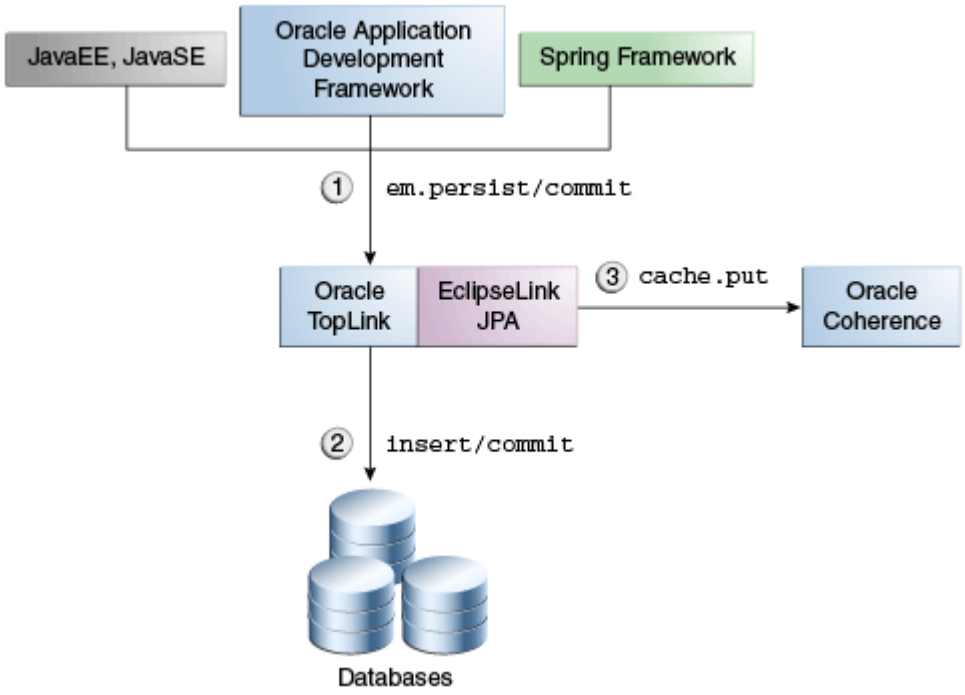
Writing Objects in Grid Cache Configuration

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 2–3 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 2–3 Writing and Persisting Objects in grid Cache Configuration



Grid Cache 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

Configuring the Cache for the Grid Cache Configuration

The cache configuration file (`coherence-cache-config.xml`) in Example 2–1 defines the cache and configures a wrapper serializer to support serialization of relationships.

Example 2–1 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>
      <!--
        Configure a wrapper serializer to support serialization of relationships.
      -->
      <serializer>
        <class-name>oracle.eclipselink.coherence.integrated.cache.
WrapperSerializer</class-name>
      </serializer>
```

```
      <backing-map-scheme>
      <!--
        Backing map scheme with no eviction policy.
      -->
        <local-scheme>
          <scheme-name>unlimited-backing-map</scheme-name>
        </local-scheme>
      </backing-map-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
```

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 2–2. This class intercepts all TopLink calls to the internal TopLink Grid cache and redirects them to the Coherence cache.

Example 2–2 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 {
...
```

Inserting Objects for the Grid Cache Configuration

In Example 2–3, 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 2–3 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();
```

Querying Objects for the Grid Cache Configuration

In Example 2–4, 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 2–4 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();
```

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 2-9.

See "Grid Read Configuration Examples" on page 2-9 for detailed examples.

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:

```
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.

```
select e.firstName, e.lastName 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.

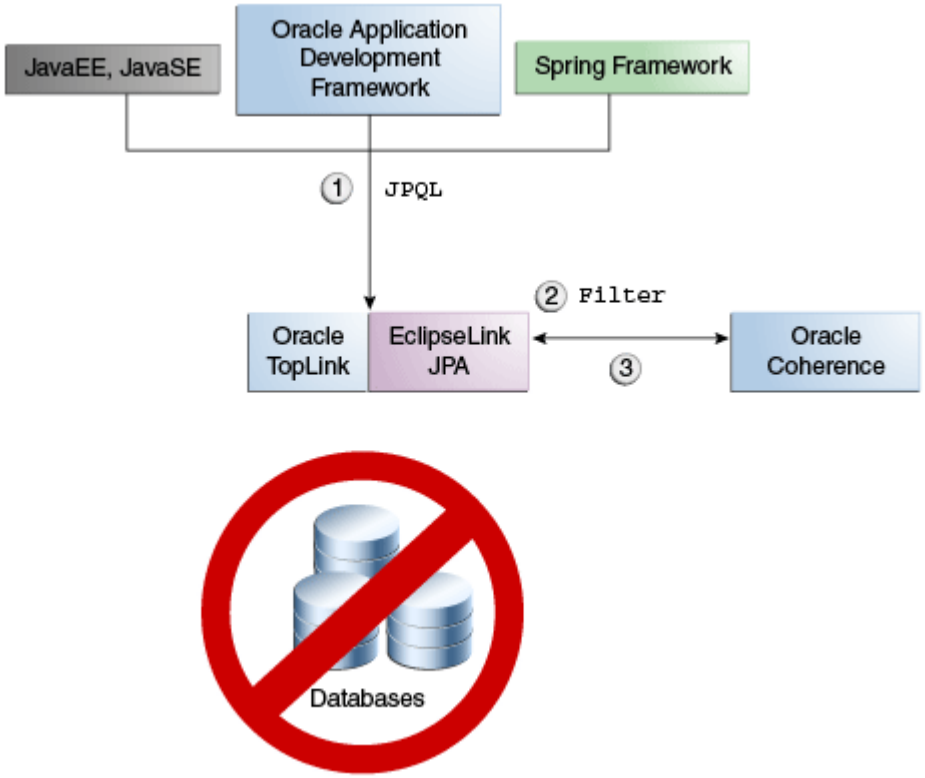
For complete information on using EclipseLink JPA query hints, see the EclipseLink documentation at this URL:

```
http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_
(ELUG)#How_to_Use_EclipseLink_JPA_Query_Hints
```

Figure 2–4 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 2–4 Reading Objects with a Query



Writing Objects in Grid Read Configuration

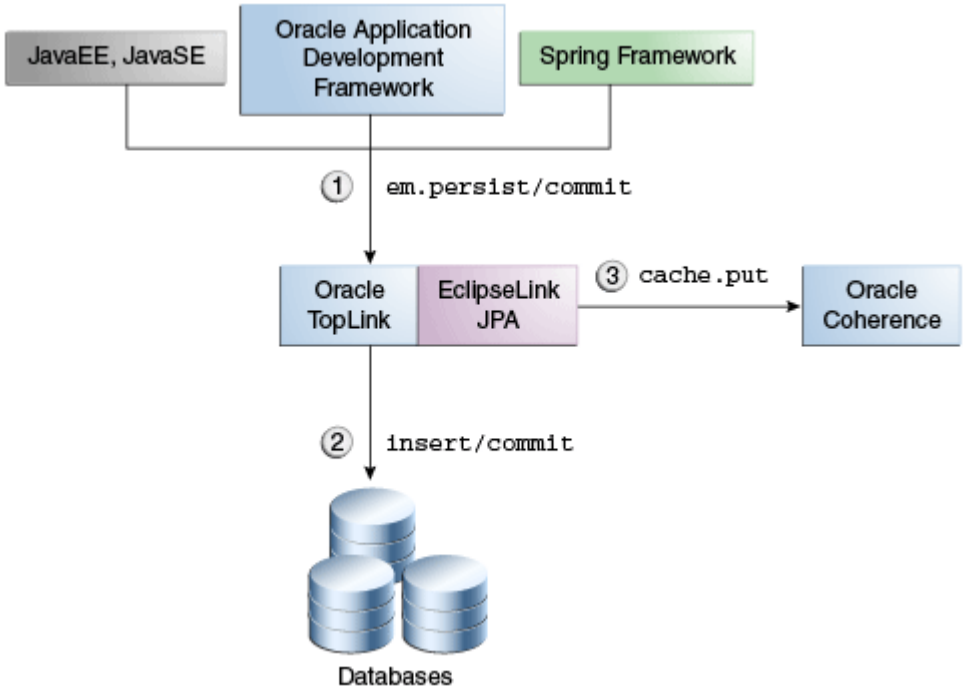
In the Grid Read configuration, TopLink performs all database write operations (insert, update, delete) directly. The Coherence caches are then updated to reflect the changes made to the database. TopLink offers a number of performance features when writing large amounts of data. These include batch 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 2–5 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 2–5 Writing and Persisting Objects in Grid Read Configuration



Grid Read 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`

Configuring the Cache in Grid Read Configuration

The cache configuration file (`coherence-cache-config.xml`) in Example 2–5 defines the cache and configures a wrapper serializer to support serialization of

relationships. The `oracle.eclipselink.coherence.integrated.EclipseLinkJPACacheLoader` class defines the cache store scheme.

Example 2–5 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>
              <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>
            </class-scheme>
          </cachestore-scheme>
          <read-only>true</readonly>
        </read-write-backing-map-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
```

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 2–6:

Example 2–6 Configuring the Entity in Grid Read Configuration

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

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

Inserting Objects for the Grid Read Configuration

In Example 2–7, 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 2–7 Inserting Objects in Grid Read 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();

emf.close();
```

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 2–8.

Example 2–8 Querying Objects in Grid Read 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();
```

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.

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 2-12.
- 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 2-12.

See "Grid Entity Configuration Examples" on page 2-13 for detailed examples.

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 2-3 for more information.

Writing Objects in Grid Entity Configuration

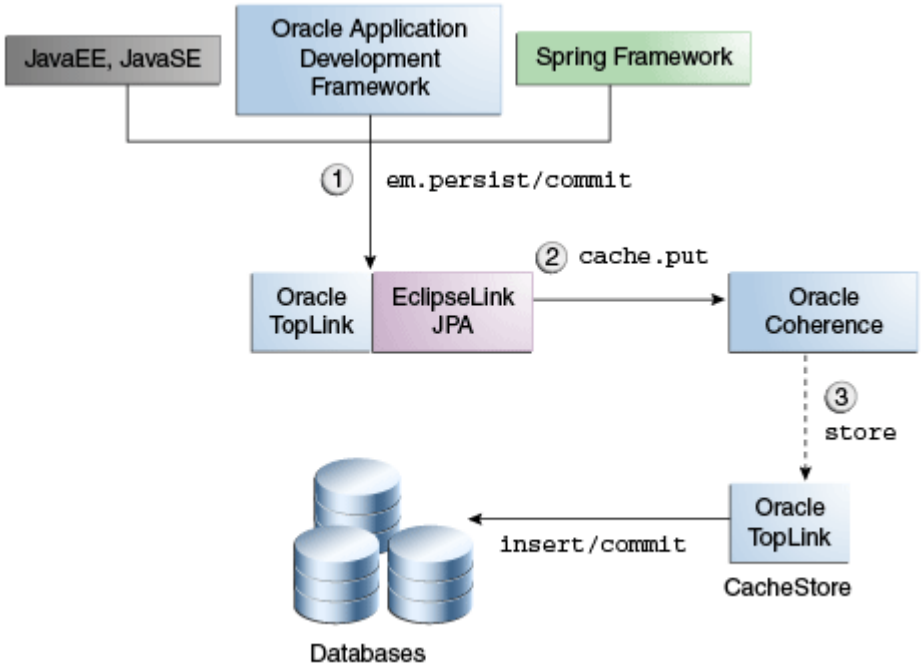
In the Grid Entity configuration, all objects that are persisted, updated, or merged through an `EntityManager` instance will be put in the appropriate Coherence cache. To persist objects in a Coherence cache to the database, an EclipseLink JPA cache store (`oracle.eclipselink.coherence.integrated.EclipseLinkJPACacheStore`)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

You can also configure the cache store to use write-behind to asynchronously batch-write updated objects. See *Coherence Developer's Guide* for more information.

Figure 2–6 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 2–6 Writing and Persisting Objects in Grid Entity Configuration



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
```

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`

Configuring the Cache for the Grid Entity Configuration

The cache configuration file (`coherence-cache-config.xml`) in Example 2–9 configures a wrapper serializer to support serialization of relationships. The `oracle.eclipselink.coherence.integrated.EclipseLinkJPACacheStore` class defines the cache store scheme.

Example 2–9 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>
      <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.
EclipseLinkJPACacheStore</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>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>
```

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 2–10:

Example 2–10 Configuring an Entity in Grid Entity Configuration

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

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

Persisting Objects for the Grid Entity Configuration

In Example 2–11, TopLink performs the insert to create a new employee. Entities persist through the `EntityManager` instance and are placed in the appropriate Coherence cache.

Example 2–11 Persisting Objects in Grid Entity 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();
```

Querying Objects for the Grid Entity Configuration

When finding an employee, the read query is directed to the Coherence cache, as shown in Example 2–12.

Example 2–12 Querying Objects in Grid Entity 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();
```

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.

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:

```
<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)`.

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 2–13. 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 2–13 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.
```

Working with Queries

This section includes information on the following topics:

- Querying Objects by ID
- Querying Objects with Criteria
- Using Indexes in Queries
- Limitations on Queries

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.

```
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.

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.

  ```
  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

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 2–14, 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 2–14 Exposing a Coherence Query Index to TopLink Grid

```
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.

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.

3

EclipseLink Native ORM Configurations

This chapter contains the following sections:

Understanding EclipseLink Native ORM

EclipseLink Native Object Relational Mapping (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.

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 3–1 describes the API for EclipseLink Native ORM. These classes can be found in the `oracle.eclipselink.coherence.integrated` package.

Table 3–1 EclipseLink Classes for Native ORM Configurations

Class Name	Description
`EclipseLinkNativeCacheStore(String cacheName, String sessionName)`	Coherence cache store that should be used with native EclipseLink configuration (`sessions.xml`).
`EclipseLinkNativeCacheLoader(String cacheName, String sessionName)`	Coherence cache loader that should be used with native EclipseLink configuration (`sessions.xml`).

Table 3–1 (Cont.) EclipseLink Classes for Native ORM Configurations

Class Name	Description
`oracle.eclipselink.coherence.integrated.config.CoherenceReadCustomizer`	Enables a Coherence read configuration.
`oracle.eclipselink.coherence.integrated.config.CoherenceReadWriteCustomizer`	Enables a Coherence read/write configuration.
`oracle.eclipselink.coherence.integrated.config.GridCacheCustomizer`	Enables entity instances to be cached in Coherence instead of in the internal EclipseLink shared cache

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 Chapter 2, "JPA on the Grid Configurations".

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.

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. How to configure amendment methods in EclipseLink Workbench is beyond the scope of this document. You can find information on this topic in "Amendment and After-Load Methods" at Eclipsepedia:

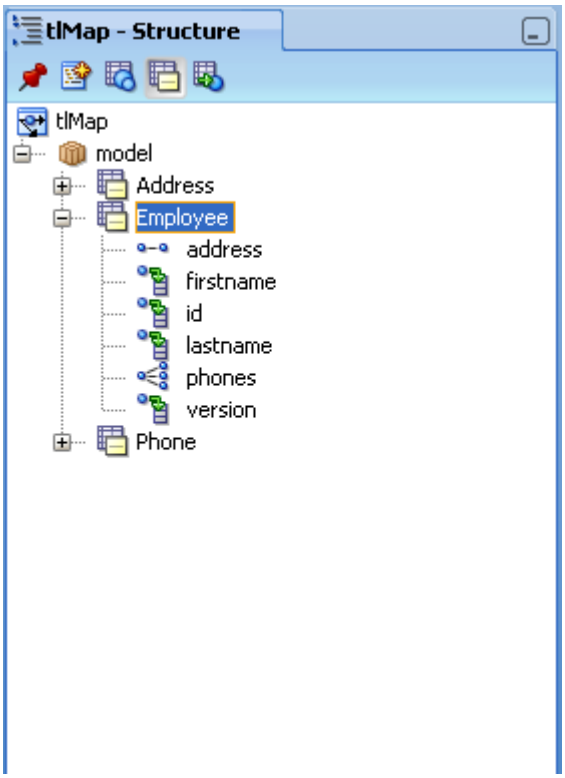
`http://wiki.eclipse.org/Introduction_to_Descriptors_(ELUG)#Amendment_and_After-Load_Methods`

Configuring the Amendment Method in JDeveloper

To configure an amendment method:

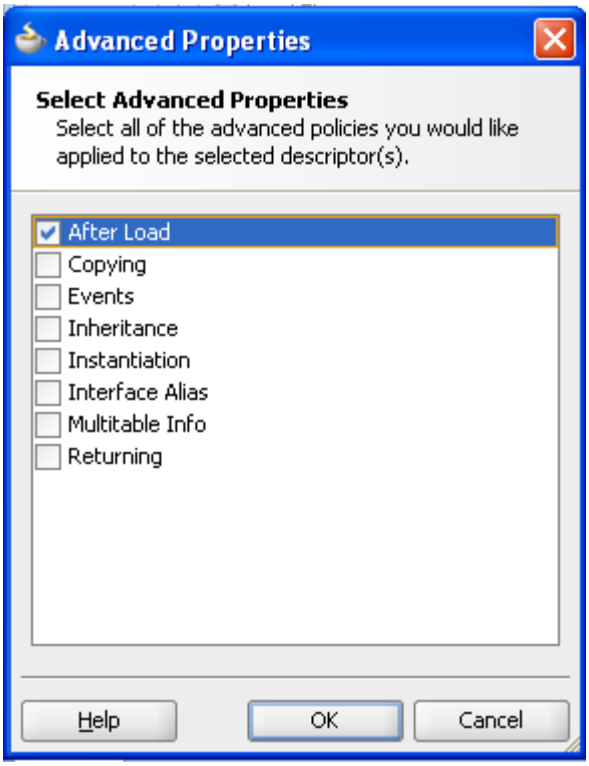
1. In the JDeveloper Structure pane, expand the desired **tlMap** descriptor name.

Figure 3–1 tlMap Descriptors in the JDeveloper Structure Pane



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**.

Figure 3–2 Advanced Properties Dialog Box



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 3–3 illustrates the After Load tab of the **tlMap** configuration window.

Figure 3–3 After Load Tab for a TopLink Descriptor

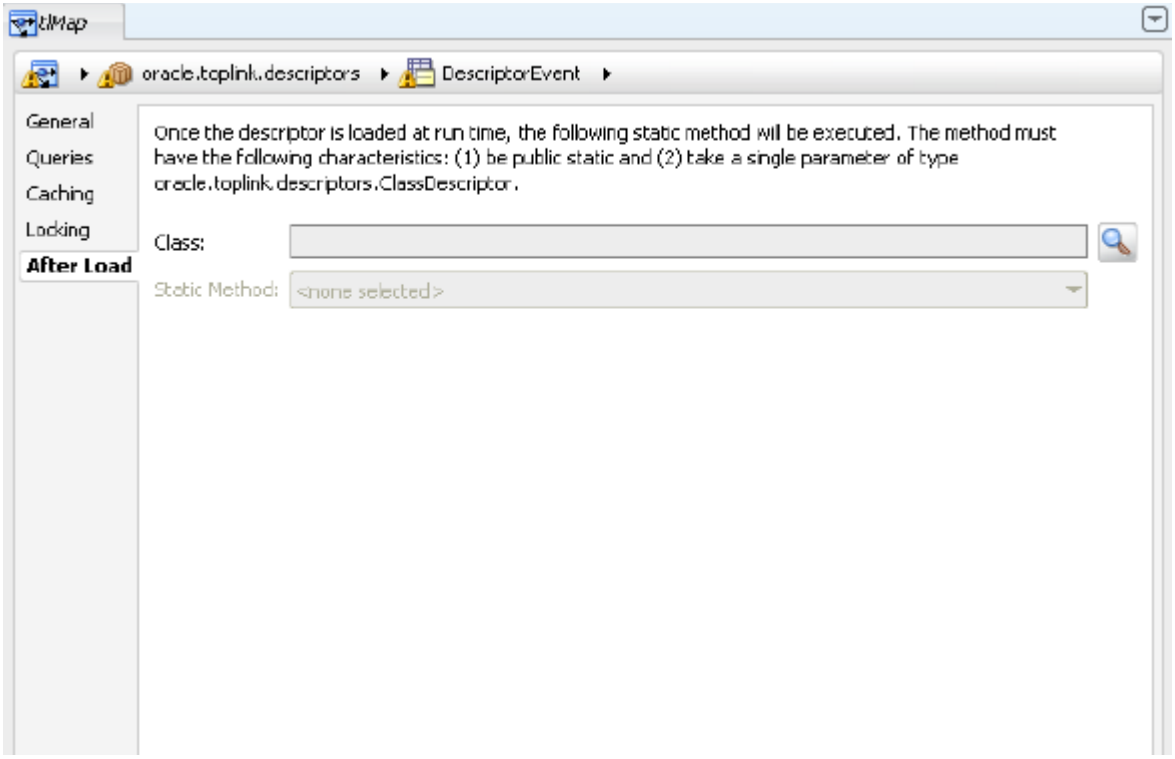
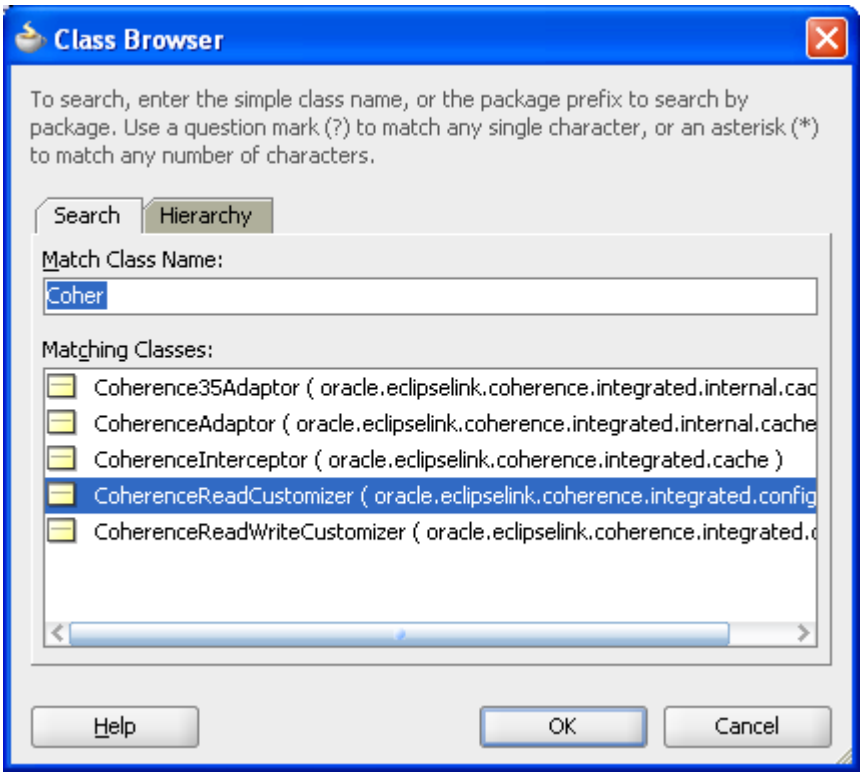


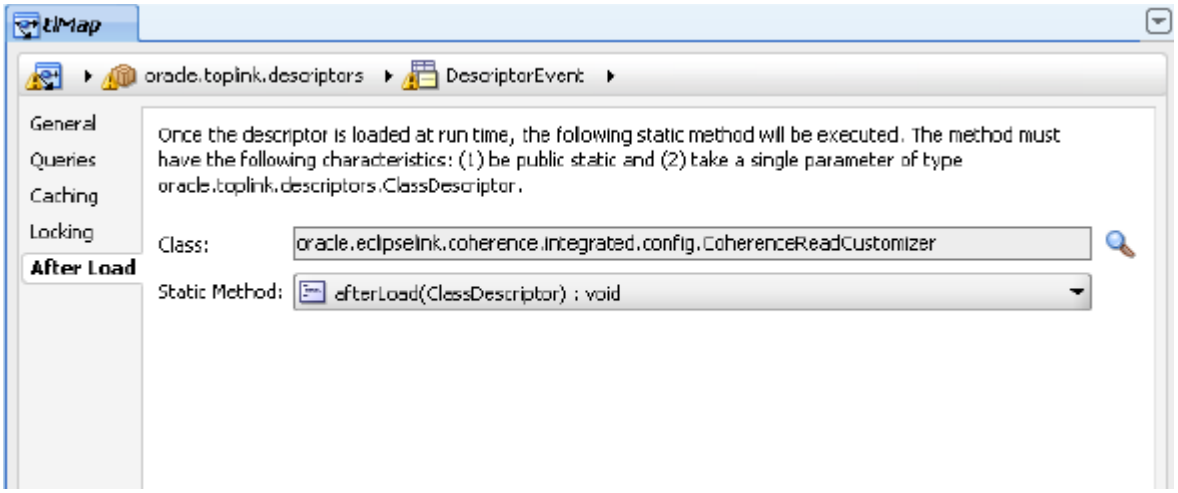
Figure 3–4 illustrates the class browser with the with the `CoherenceReadCustomizer` class selected.

Figure 3–4 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 3–5 Selecting the Amendment Method



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 3–1 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 3–1 Configuration for an Integrated EclipseLinkNativeCacheStore

```
...
<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>
            <local-scheme/>
          </internal-cache-scheme>
          <!-- Define the cache scheme -->
          <cachestore-scheme>
            <class-scheme>
              <class-name>oracle.eclipselink.coherence.integrated.
EclipseLinkNativeCacheStore</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>
        </read-write-backing-map-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
    </distributed-scheme>
...
```

Example 3–2 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 3–2 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>
...
```

4

Using POF Serialization

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 chapter 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 *Developers Guide for Oracle Coherence*.

This chapter contains the following sections:

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 4–1 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. *Do not* serialize or deserialize relationship mappings; TopLink Grid will perform these operations automatically.

Example 4–1 Sample Entity Class that Implements PortableObject

```
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)`

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 4–2 Sample Cache Configuration File

```
<?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 http://xmlns.oracle.
com/coherence/coherence-cache-config/1.0/coherence-cache-config.xsd">
  <caching-scheme-mapping>
    <cache-mapping>
        <cache-name>ATTORNEY_JPA_CACHE</cache-name>
        <scheme-name>eclipselink-jpa-distributed</scheme-name>
    </cache-mapping>
    <cache-mapping>
        <cache-name>CONTACT_JPA_CACHE</cache-name>
        <scheme-name>eclipselink-jpa-distributed-load</scheme-name>
    </cache-mapping>
...
additional cache mappings
...
<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>
      <backing-map-scheme>
        <read-write-backing-map-scheme>
          <internal-cache-scheme>
            <local-scheme/>
          </internal-cache-scheme>
        </read-write-backing-map-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
    </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>

      <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.
EclipseLinkJPACacheStore</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-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>
```

Provide 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 `pof-config.xml` by default. Use the file to assign `type-ids` to the Entity classes.

The POF configuration file must also contain `type-id` entries for the following classes:

- `oracle.eclipselink.coherence.integrated.internal.cache.WrapperInternal`—This interface is used to access internal attributes of the Entity wrappers.

  `oracle.eclipselink.coherence.integrated.cache.WrapperPofSerializer`—Associated serializer. This class is used to provide serialization support for the Entity Wrappers within Coherence when you want to access Coherence caches directly. This includes users who have custom Value Extractors.

- `oracle.eclipselink.coherence.integrated.internal.querying.EclipseLinkExtractor`—This interface is used for Coherence POF serialization to mark an EclipseLink Extractor for serialization. It extracts values from TopLink Grid entities for Filters.

  `oracle.eclipselink.coherence.integrated.cache.ExtractorSerializer`—Associated serializer. This class is used to provide serialization support for the Entity Wrappers within Coherence when you want to access the Coherence caches directly. This includes users who have custom Value Extractors.

- `oracle.eclipselink.coherence.integrated.internal.cache.VersionPutProcessor`—An internal file, used for optimistic lock-aware updates to the grid.

- `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.RelationshipUpdateProcessor`—An internal file, used to update lazy-loaded relationship data into the Grid.

- `oracle.eclipselink.coherence.integrated.internal.querying.EclipseLinkFilterFactory$SubClassOf`—An internal file. This is a Filter extension that filters on the type of Entity, eliminating superclasses from polymorphic queries.

Example 4–3 illustrates a sample POF configuration file that includes the TopLink Grid support files.

Example 4–3 Sample POF Configuration File

```
<?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
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>1144</type-id>
       <class-name>oracle.eclipselink.coherence.integrated.internal.cache.
WrapperInternal</class-name>
       <serializer>
         <class-name>oracle.eclipselink.coherence.integrated.cache.
WrapperPofSerializer</class-name>
       </serializer>
     </user-type>
     <user-type>
       <type-id>1142</type-id>
       <class-name>oracle.eclipselink.coherence.integrated.internal.querying.
EclipseLinkExtractor</class-name>
       <serializer>
         <class-name>oracle.eclipselink.coherence.integrated.cache.
ExtractorSerializer</class-name>
       </serializer>
     </user-type>
     <user-type>
       <type-id>1141</type-id>
       <class-name>oracle.eclipselink.coherence.integrated.internal.cache.
VersionPutProcessor</class-name>
     </user-type>
     <user-type>
       <type-id>1140</type-id>
       <class-name>oracle.eclipselink.coherence.integrated.internal.cache.
VersionRemoveProcessor</class-name>
     </user-type>
     <user-type>
       <type-id>1139</type-id>
       <class-name>oracle.eclipselink.coherence.integrated.internal.cache.
RelationshipUpdateProcessor</class-name>
     </user-type>
     <user-type>
       <type-id>1138</type-id>
       <class-name>oracle.eclipselink.coherence.integrated.internal.querying.
EclipseLinkFilterFactory$SubClassOf</class-name>
     </user-type>
   </user-type-list>

   <allow-interfaces>true</allow-interfaces>
   <allow-subclasses>true</allow-subclasses>
 </pof-config>
```

5

Best Practices

This chapter contains recommendations of how to use TopLink Grid with byte code weaving and lazy loading:

- 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

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 "How to Configure Dynamic Weaving for JPA Entities Using the EclipseLink Agent" at the following URL for more information about dynamic byte code weaving for JRE 1.5.

`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG)#How_to_Configure_Dynamic_Weaving_for_JPA_Entities_Using_the_EclipseLink_Agent`

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 EclipseLink JPA Weaving" at Eclipsepedia for more information on configuring and disabling static and dynamic byte code weaving.

`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG)#Using_EclipseLink_JPA_Weaving`

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:

```
...
@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 5-1.

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 *Developers Guide for Oracle Coherence* for more information on near caches.

Note: Near caches are used only on a Coherence cache `get` operation, but not when a `Filter` operation is executed. This is because the `Filter` operation is sent to each member, and they return results directly to the caller. In this case, a near cache will not add value.

This can also become an issue if you are using JPQL queries. In the TopLink Grid Grid Read or Grid Entity configurations, JPQL queries are mapped to `Filter` operations. In the case of either of these configurations, if you execute TopLink JPQL queries, you will not see any cache hits.

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 5–1 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 5–1 Session Customizer to Prepend

```
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.getCacheInterceptorClass())) {
            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>
```

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.

Index

Symbols

A

B

C

D

E

F

G

I

J

L

N

O

P

Q

S

T

W