Oracle® Coherence

Integration Guide for Oracle Coherence

Release 3.6

E15830-01

July 2010

ORACLE®

Oracle Coherence Integration Guide for Oracle Coherence, Release 3.6

E15830-01

Primary Author: Thomas Pfaeffle

Contributing Author: Noah Arliss, Jason Howes, Mark Falco, Alex Gleyzer, Gene Gleyzer, David Leibs, Tim Middleton, Andy Nguyen, Brian Oliver, Patrick Peralta, Cameron Purdy, Jonathan Purdy, Everet Williams, Tom Beerbower, John Speidel

Contents

List of Examples

List of Tables

Preface

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 Hibernate, and Spring, and EclipseLink Essentials.

Documentation Accessibility

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Related Documents

For more information about Oracle Coherence, see the following:

- *Getting Started for Oracle Coherence*
- *Developer's Guide for Oracle Coherence*
- *Client Guide for Oracle Coherence*
- *Tutorial for Oracle Coherence*
- *User's Guide for Oracle Coherence*Web*

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

Configuring Coherence for JPA

The Java Persistence API (JPA) is the standard for object-relational mapping (ORM) and enterprise Java persistence. This chapter describes the following:

- How to use the default, entity-based Coherence implementations of the cache store and cache loader. These implementations use JPA to load and store objects to the database.
- How to use the Coherence API with caches backed by TopLink Grid to access relational data with JPA cache loader and cache store implementations optimized for EclipseLink JPA.

> **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 also provides a brief introduction to *JPA on the Grid*, where JPA applications use TopLink Grid to interact directly with the database. TopLink Grid, in turn, uses the Coherence data grid to store some or all of the domain model. For a detailed discussion of *JPA on the Grid*, see the *Integration Guide for Oracle TopLink with Coherence Grid*.

This chapter contains the following sections:

API for Native Coherence and Coherence JPA CacheStore and CacheLoader

Oracle Coherence provides its own implementations of the `CacheLoader` and `CacheStore` classes which can be used with JPA. The `JpaCacheLoader` and `JpaCacheStore` classes 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 called `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.

Table 1–1 describes the default JPA implementations provided by Coherence.

Table 1–1 JPA-Related CacheStore and CacheLoader API Included with Coherence

Class Name	Description
`com.tangosol.net.cache.CacheLoader`	A JCache cache loader.
`com.tangosol.net.cache.CacheStore`	A JCache cache store. The `CacheStore` interface extends `CacheLoader`.
`com.tangosol.coherence.jpa.JpaCacheLoader`	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.
`com.tangosol.coherence.jpa.JpaCacheStore`	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. **Note:** The persistence unit is assumed to be set to use `RESOURCE_LOCAL` transactions.

Using the Default Coherence JPA

To use Coherence JPA 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.

Obtain a JPA Provider Implementation

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

A JPA provider is not included in the Coherence distribution, but you can obtain one. Although the Coherence JPA cache store works with any JPA-compliant implementation, Oracle recommends using EclipseLink JPA, the reference implementation for the JPA 2.0 specification. EclipseLink JPA is available from Eclipse at the following URL:

`http://www.eclipse.org/eclipselink`

Oracle TopLink and TopLink Grid for Coherence integration include EclipseLink as their JPA implementations. For more information about TopLink and to download it, go to the following URL:

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

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. TopLink 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

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.

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 1–1 illustrates a sample `persistence.xml` file with the typical properties that you can set.

Example 1–1 Sample persistence.xml File for JPA

```
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance" 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>
```

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 1–2 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 1–2 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 1–2 Assigning Named Caches to a JPA Caching Scheme

```
<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>
  <caching-schemes>
    <distributed-scheme>
      <scheme-name>jpa-distributed</scheme-name>
      <service-name>JpaDistributedCache</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>
              <class-name>
                com.tangosol.coherence.jpa.JpaCacheStore
              </class-name>
              <init-params>

                <!-- This param is the entity name. -->
                <init-param>
                  <param-type>java.lang.String</param-type>
                  <param-value>{cache-name}</param-value>
```

```
                </init-param>

                <!-- This param is the fully qualified entity class. -->
                <init-param>
                  <param-type>java.lang.String</param-type>
                  <param-value>com.acme.{cache-name}</param-value>
                </init-param>

                <!-- This param should match the value of the -->
                <!-- persistence unit name in persistence.xml. -->
                <init-param>
                  <param-type>java.lang.String</param-type>
                  <param-value>EmpUnit</param-value>
                </init-param>
              </init-params>
            </class-scheme>
          </cachestore-scheme>
        </read-write-backing-map-scheme>
      </backing-map-scheme>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
```

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 *Read-Through, Write-Through, Write-Behind, and Refresh-Ahead Caching* in *Oracle Coherence Getting Started Guide* and *Sample CacheStore* in *Oracle Coherence Developer's Guide* 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.

Using Coherence Caches Backed by TopLink Grid

Figure 1–1 illustrates the relationship between the client application (which employs Coherence APIs), the Coherence cache, TopLink Grid, and the database.

Figure 1–1 Coherence with TopLink Grid Approach

API for Coherence with TopLink Grid Configurations

The TopLink Grid cache store and cache loader implementations are shipped in the `toplink-grid.jar` file. 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 classes which are optimized for EclipseLink JPA and designed for use by Coherence applications, are in the `oracle.eclipselink.coherence.standalone` package. Table 1–2 describes these classes.

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

Class Name	Description
`EclipseLinkJPACacheLoader`	Provides JPA-aware versions of the Coherence `CacheLoader` class.
`EclipseLinkJPACacheStore`	Provides JPA-aware versions of the Coherence `CacheStore` class.

Examples of Coherence with TopLink Grid Configurations

In the cache configuration (`coherence-cache-config.xml`) define the cache, as illustrated in Example 1–3. 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 1–3, 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 1–3, `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 1–3 Configuring the Cache for Coherence with TopLink Grid

```
<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>
```

Using Coherence as the TopLink Grid Cache

> **Note:** This section provides only an introduction to using Coherence as the cache for TopLink Grid. See *Integration Guide for Oracle TopLink with Coherence Grid* for more information.

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.

These are the typical TopLink Grid configurations that applications can use:

- Coherence Shared L2 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.

- Coherence 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 would be used to populate the Coherence cache, but individual queries could be directed to the database if necessary.

- Coherence Read/Write 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.

See the Oracle TopLink Grid page on the Oracle Technology Network for more information about configuring Coherence for TopLink Grid.

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

2

Integrating Hibernate and Oracle Coherence

Hibernate is 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.

This chapter contains the following sections:

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 2–1 describes the different constructors for the `HibernateCacheStore` and `HibernateCacheLoader` classes. For more detailed technical information, see the Javadoc for these classes.

Table 2–1 Hibernate Cache Store and Cache Loader Classes

Class Name	Description
`HibernateCacheLoader()` and `HibernateCacheStore()`	These constructors are the default constructors for creating a new instance of a cache loader or cache store. They do not create a Hibernate `SessionFactory` object. To create a Hibernate `SessionFactory` object when you are using these constructors, use the `setSession()` method.
`HibernateCacheLoader(java.lang.String entityName)` and `HibernateCacheStore(java.lang.String entityName),`	These constructors create a Hibernate `SessionFactory` object using the default Hibernate configuration (`hibernate.cfg.xml`) in the classpath.

Table 2–1 (Cont.) Hibernate Cache Store and Cache Loader Classes

Class Name	Description
`HibernateCacheStore(java.lang.String entityName, java.lang.String sResource)` and `HibernateCacheStore(java.lang.String entityName, java.lang.String sResource)`	These constructors create a Hibernate `SessionFactory` object based on the configuration file provided (`sResource`).
`HibernateCacheLoader(java.lang.String entityName, java.io.File configurationFile)` and `HibernateCacheStore(java.lang.String entityName, java.io.File configurationFile),`	These constructors create a Hibernate `SessionFactory` object based on the configuration file provided (`configurationFile`)
`HibernateCacheStore(java.lang.String entityName, org.hibernate.SessionFactory sFactory)` and `HibernateCacheStore(java.lang.String entityName, org.hibernate.SessionFactory sFactory),`	These constructors accept an `entityName` name and a Hibernate `SessionFactory`.

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 adavantage of custom CacheStore implementations)
- Have simpler transactional requirements
- Require high performance through the use of Coherence APIs, such as write-behind and aggregations.

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.

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 `hibernate.cfg.xml` file as the second `<init-param>` (set the `<param-type>` element to `java.lang.String` for a resource name and `java.io.File` for a file specification). See the Javadoc for `HibernateCacheStore` for more information.

Example 2–1 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 2–1 Coherence Cache Configuration File for Hibernate

```
<?xml version="1.0"?>

<!DOCTYPE cache-config SYSTEM "cache-config.dtd">

<cache-config>
  <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></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>{entityname}</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 2–2 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 2–2 Coherence Cache Configuration File that Uses {cache-name} Macro

```
<?xml version="1.0"?>

<!DOCTYPE cache-config SYSTEM "cache-config.dtd">
```

```
<cache-config>
  <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 2–3 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 2–3 Sample coherence-cache-config.xml File with Generalized Mappings

```
<?xml version="1.0"?>

<!DOCTYPE cache-config SYSTEM "cache-config.dtd">

<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>com.company.*</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>{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>
```

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.

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 programmatically or by using the `hibernate.cfg.xml` file), with `CacheStore` implementations using the version with the cache disabled.

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, programmatically configured `SessionFactory` instance.

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.

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).

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.

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.

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.

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.

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 2–4 illustrates the `property` element calling `CoherenceCacheProvider` (`com.tangosol.coherence.hibernate.CacheProvider`) as the value of the `hibernate.cache.provider_class` property.

Example 2–4 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 opearations. 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.

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.

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.

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.

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).

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).

3

Integrating Oracle Coherence with Spring

Spring is a platform for building and running Java EE applications. This chapter describes how to configure the Oracle Coherence cache to be available to applications that run on the Spring platform.

Coherence `CacheFactory` static factory methods allow you to access all Coherence caches and services. These methods, (such as `getCache`), delegate to a `ConfigurableCacheFactory` interface, which can be plugged in by using the `CacheFactory.setConfigurableCacheFactory` method or the operational override file (`tangosol-coherence-override.xml`).

The Coherence cache configuration file (`coherence-cache-config.xml`), provides the `class-scheme` element, where you can specify your own implementations of Coherence interfaces, such as `CacheStore` and `MapListener`. Coherence can instantiate your implementatons in either of two ways: it can create a new instance by using the new operator, or it can invoke a user-provided factory method.

For some applications, it may be useful for Coherence to retrieve objects configured in a `class-scheme` element from a Spring `BeanFactory` instance instead of creating its own instance. This is especially true for cache servers configured with `CacheStore` objects running in a standalone JVM, because these `CacheStore` objects typically must be configured with data sources, connection pools, and so on. Spring provides easy configuration of data sources for plain Java objects.

The `SpringAwareCacheFactory` interface is a custom `ConfigurableCacheFactory` that can delegate class scheme bean instantiations to a Spring `BeanFactory` instance. It has two modes of operation:

- It can instantiate its own `ApplicationContext` instance with a provided configuration file. This is useful for cache servers that require beans from a Spring container.
- A `BeanFactory` instance can be provided to it at run time. This is useful for Coherence applications running in a container that already has an existing `BeanFactory` instance.

Configuring Coherence for a Spring-Aware Cache Factory

To configure Coherence to use the `SpringAwareCacheFactory` instance, the XML code in Example 3–1 should be placed in the operational override file (`tangosol-coherence-override.xml`). By default, this specifies `coherence-cache-config.xml` as the cache configuration file and `application-context.xml` as the Spring configuration file.

Example 3–1 Configuring the Cache to Use SpringAwareCacheFactory

```
...
<configurable-cache-factory-config>
  <class-name system-property="tangosol.coherence.cachefactory">
    com.tangosol.coherence.spring.SpringAwareCacheFactory
  </class-name>
  <init-params>
    <init-param>
      <param-type>java.lang.String</param-type>
      <param-value system-property="tangosol.coherence.cacheconfig">
         coherence-cache-config.xml
      </param-value>
    </init-param>
    <init-param id="1">
      <param-type>java.lang.String</param-type>
      <param-value system-property="tangosol.coherence.springconfig">
         application-context.xml
      </param-value>
    </init-param>
  </init-params>
</configurable-cache-factory-config>
...
```

As an alternative to using the configuration file, the `SpringAwareCacheFactory` instance can be configured programmatically as illustrated in Example 3–2:

Example 3–2 Configuring SpringAwareCacheFactory Programmatically

```
BeanFactory            bf  = ...
SpringAwareCacheFactory scf = new SpringAwareCacheFactory();

scf.setBeanFactory(bf);
CacheFactory.setConfigurableCacheFactory(scf);
```

Because the `SpringAwareCacheFactory` instance is `BeanFactoryAware`, it can also be defined in an application context, as shown in Example 3–3:

Example 3–3 Defining a SpringAwareCacheFactory Instance in an Application Context

```
<bean id="cacheFactory"
      class="com.tangosol.coherence.spring.SpringAwareCacheFactory">
</bean>
```

Taking this a step further, the Coherence `CacheFactory` instance can be configured inside of the application context, as shown in Example 3–4:

Example 3–4 Configuring a CacheFactory Instance in an Application Context

```
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetClass" value="com.tangosol.net.CacheFactory"/>
  <property name="targetMethod" value="setConfigurableCacheFactory"/>
  <property name="arguments" ref="cacheFactory"/>
</bean>
```

The application context can have a `CacheStore` instance configured as in Example 3–5. Note that the `EntityCacheStore` instance is scoped as `prototype`. This value is specified because Coherence will manage the life cycle of the bean when it is retrieved from Spring, just as if Coherence had instantiated the object using the `new` operator.

Example 3–5 Configuring a CacheStore Instance in an Application Context

```
<bean id="dataSource" class="...">
...
</bean>

<bean id="sessionFactory" class="...">
  <property name="dataSource" ref="dataSource"/>
  ...
</bean>

<bean id="entityCacheStore"
      class="com.company.app.EntityCacheStore" scope="prototype">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Coherence can use the `entityCacheStore` bean as illustrated in Example 3–6. By specifying the `init-param` element, setter injection can be used to set properties on the bean retrieved from Spring. The bean will have the method `setEntityName` invoked with the cache name before it is used by Coherence.

Example 3–6 Configuring Setter Injection to Set Properties on the Bean

```
<?xml version="1.0"?>

<!DOCTYPE cache-config SYSTEM "cache-config.dtd">

<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>com.company.app.domain.*</cache-name>
      <scheme-name>distributed-domain</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>

  <caching-schemes>
    <distributed-scheme>
      <scheme-name>distributed-domain</scheme-name>
      <backing-map-scheme>
        <read-write-backing-map-scheme>
          <internal-cache-scheme>
            <local-scheme />
          </internal-cache-scheme>
          <cachestore-scheme>
            <class-scheme>
              <class-name>spring-bean:entityCacheStore</class-name>
              <init-params>
                <init-param>
                  <param-name>setEntityName</param-name>
                  <param-value>{cache-name}</param-value>
                </init-param>
              </init-params>
            </class-scheme>
          </cachestore-scheme>
          <write-delay>5s</write-delay>
        </read-write-backing-map-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
```

SpringAwareCacheFactory API

Example 3–7 lists the Java source code for `SpringAwareCacheFactory`. It requires Coherence 3.4.*n* and Spring 2.*n*.

Example 3–7 Updated Sample for a Spring-Aware Cache Factory

```
package com.tangosol.coherence.spring;

import com.tangosol.net.BackingMapManagerContext;
import com.tangosol.net.DefaultConfigurableCacheFactory;

import com.tangosol.run.xml.SimpleElement;
import com.tangosol.run.xml.XmlElement;
import com.tangosol.run.xml.XmlHelper;

import com.tangosol.util.ClassHelper;

import java.util.Iterator;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;


/**
* SpringAwareCacheFactory provides a facility to access caches declared
* in a "cache-config.dtd" compliant configuration file, similar to its super
* class {@link DefaultConfigurableCacheFactory}.  In addition, this factory
* provides the ability to reference beans in a Spring application context
* through the use of a class-scheme element.
*
* This factory can be configured to start its own Spring application
* context from which to retrieve these beans.  This can be useful for standalone
* JVMs such as cache servers.  It can also be configured at run time with a
* preconfigured Spring bean factory.  This can be useful for Coherence
* applications running in an environment that is itself responsible for starting
* the Spring bean factory, such as a web container.
*
* @see #instantiateAny(CacheInfo, XmlElement,
       BackingMapManagerContext, ClassLoader)
*
*/
public class SpringAwareCacheFactory
        extends DefaultConfigurableCacheFactory
        implements BeanFactoryAware
    {
    // ----- constructors ---------------------------------------------------

    /**
    * Construct a default DefaultConfigurableCacheFactory using the
    * default configuration file name.
    */
    public SpringAwareCacheFactory()
        {
        super();
        }
```

```
/**
* Construct a SpringAwareCacheFactory using the specified path to
* a "cache-config.dtd" compliant configuration file or resource.  This
* will also create a Spring ApplicationContext based on the supplied
* path to a Spring compliant configuration file or resource.
*
* @param sCacheConfig location of a cache configuration
* @param sAppContext  location of a Spring application context
*/
public SpringAwareCacheFactory(String sCacheConfig, String sAppContext)
    {
    super(sCacheConfig);

    azzert(sAppContext != null && sAppContext.length() > 0,
            "Application context location required");

    m_beanFactory = sCacheConfig.startsWith("file:") ? (BeanFactory)
        new FileSystemXmlApplicationContext(sAppContext) :
        new ClassPathXmlApplicationContext(sAppContext);

    // register a shutdown hook so the bean factory cleans up
    // upon JVM exit
    ((AbstractApplicationContext) m_beanFactory).registerShutdownHook();
    }

/**
* Construct a SpringAwareCacheFactory using the specified path to
* a "cache-config.dtd" compliant configuration file or resource and
* the supplied Spring BeanFactory.
*
* @param sPath       the configuration resource name or file path
* @param beanFactory Spring BeanFactory used to load Spring beans
*/
public SpringAwareCacheFactory(String sPath, BeanFactory beanFactory)
    {
    super(sPath);

    m_beanFactory = beanFactory;
    }


// ----- extended methods ---------------------------------------------

/**
* Create an Object using the "class-scheme" element.
*
* In addition to the functionality provided by the super class,
* this will retreive an object from the configured Spring BeanFactory
* for class names that use the following format:
*
* &lt;class-name&gt;spring-bean:sampleCacheStore&lt;/class-name&gt;
*
*
* Parameters may be passed to these beans through setter injection as well:
*
*   &lt;init-params&gt;
*     &lt;init-param&gt;
*       &lt;param-name&gt;setEntityName&lt;/param-name&gt;
*       &lt;param-value&gt;{cache-name}&lt;/param-value&gt;
```

```
     *     &lt;/init-param&gt;
     *   &lt;/init-params&gt;
     *
     *
     * Note that Coherence will manage the lifecycle of the instantiated Spring
     * bean, therefore any beans that are retrieved using this method should be
     * scoped as a prototype in the Spring configuration file, for example:
     *
     *   &lt;bean id="sampleCacheStore"
     *         class="com.company.SampleCacheStore"
     *         scope="prototype"/&gt;
     *
     *
     * @param info      the cache info
     * @param xmlClass  "class-scheme" element.
     * @param context   BackingMapManagerContext to be used
     * @param loader    the ClassLoader to instantiate necessary classes
     *
     * @return a newly instantiated Object
     *
     * @see DefaultConfigurableCacheFactory#instantiateAny(
     *        CacheInfo, XmlElement, BackingMapManagerContext, ClassLoader)
     */
    public Object instantiateAny(CacheInfo info, XmlElement xmlClass,
            BackingMapManagerContext context, ClassLoader loader)
        {
        if (translateSchemeType(xmlClass.getName()) != SCHEME_CLASS)
            {
            throw new IllegalArgumentException(
                    "Invalid class definition: " + xmlClass);
            }

        String sClass = xmlClass.getSafeElement("class-name").getString();

        if (sClass.startsWith(SPRING_BEAN_PREFIX))
            {
            String sBeanName = sClass.substring(SPRING_BEAN_PREFIX.length());

            azzert(sBeanName != null && sBeanName.length() > 0,
                    "Bean name required");

            XmlElement xmlParams = xmlClass.getElement("init-params");
            XmlElement xmlConfig = null;
            if (xmlParams != null)
                {
                xmlConfig = new SimpleElement("config");
                XmlHelper.transformInitParams(xmlConfig, xmlParams);
                }

            Object oBean = getBeanFactory().getBean(sBeanName);

            if (xmlConfig != null)
                {
                for (Iterator iter = xmlConfig.getElementList().iterator(); iter.
hasNext();)
                    {
                    XmlElement xmlElement = (XmlElement) iter.next();

                    String sMethod = xmlElement.getName();
                    String sParam  = xmlElement.getString();
```

```
                try
                    {
                    ClassHelper.invoke(oBean, sMethod, new Object[]{sParam});
                    }
                catch (Exception e)
                    {
                    ensureRuntimeException(e,"Could not invoke " + sMethod +
                            "(" + sParam + ") on bean " + oBean);
                    }
                }
            }
        return oBean;
        }
    else
        {
        return super.instantiateAny(info, xmlClass, context, loader);
        }
    }

/**
* Get the Spring BeanFactory used by this CacheFactory.
* @return the Spring {@link BeanFactory} used by this CacheFactory
*/
public BeanFactory getBeanFactory()
    {
    azzert(m_beanFactory != null, "Spring BeanFactory == null");
    return m_beanFactory;
    }

/**
* Set the Spring BeanFactory used by this CacheFactory.
* @param beanFactory the Spring {@link BeanFactory} used by this CacheFactory
*/
public void setBeanFactory(BeanFactory beanFactory)
    {
    m_beanFactory = beanFactory;
    }


// ----- data fields ----------------------------------------------------

/**
* Spring BeanFactory used by this CacheFactory
*/
private BeanFactory m_beanFactory;

/**
* Prefix used in cache configuration "class-name" element to indicate
* this bean is in Spring.
*/
private static final String SPRING_BEAN_PREFIX = "spring-bean:";
}
```

4

Integrating WebLogic Portal and Oracle Coherence

Oracle Coherence integrates closely with Oracle WebLogic Portal. Specifically, Coherence includes the following integration points:

- Coherence*Web for HTTP session state management
- Personalization (P13N) CacheProvider SPI implementation
- A blueprint for efficiently sharing data between Web Services for Remote Portlets (WSRP)-federated portals that uses Coherence and the WebLogic Portal Custom Data Transfer mechanism

This chapter contains the following sections:

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:

`http://download.oracle.com/docs/cd/E13155_01/wlp/docs103/javadoc/com/bea/p13n/cache/package-summary.html`

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 exceding 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:

   ```
   <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/E13155_01/wlp/docs103/caches/caches.html`

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 on WSRP Federated Portals and Coherence, see the following URL:

`http://www.oracle.com/technetwork/articles/entarch/federated-portal-cache-090433.html`

Index

T

W