Oracle® Coherence

Integrating Oracle Coherence

14.1.1.2206
F44673-06
July 2024

ORACLE

Oracle Coherence Integrating Oracle Coherence, 14.1.1.2206

F44673-06

Primary Author: Oracle Corporation

Contents

Preface

Integrating Oracle Coherence describes how to integrate Oracle Coherence with Coherence*Web, EclipseLink JPA, Hibernate, Spring, memcached adapters, and Coherence GoldenGate HotCache.

This preface includes the following sections:

Audience

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

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `https://www.oracle.com/corporate/accessibility/`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `https://support.oracle.com/portal/` or visit `Oracle Accessibility Learning and Support` if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

For more information about Oracle Coherence, see the following:

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

Conventions

The following text conventions are used in this document:

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

Using JPA with Coherence

Coherence provides native, entity-based implementations of the `CacheStore` and `CacheLoader` interfaces that use the Java Persistence API (JPA) to load and store objects to a database. Before using JPA with Coherence, you should be familiar with the `CacheStore` and `CacheLoader` interfaces. These interfaces are used to cache data sources. See Caching Data Sources.

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

This chapter includes the following sections:

- Overview of the JPA CacheStore and CacheLoader Implementations
- Obtaining a JPA Provider Implementation
  A JPA provider allows you to work directly with Java objects, rather then with SQL statements. You map, store, update and retrieve data, and the provider performs the translation between database entities and Java objects.
- Configuring a Coherence JPA Cache Store
  Using JPA with Coherence requires configuring persistence properties and defining a cache that uses the `JpaCacheStore` implementation.

Overview of the JPA CacheStore and CacheLoader Implementations

Oracle Coherence provides two implementations of the `CacheStore` and `CacheLoader` interfaces which can be used with JPA: a generic JPA implementation and an EclipseLink-specific implementation. For both implementations, 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. The JPA run-time configuration file, `persistence.xml`, and the default JPA Object-Relational mapping file, `orm.xml`, are typically provided as part of a JPA solution.
Table 1-1 describes the JPA implementations provided by Coherence.

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

Class Name	Location	Description
`JpaCacheStore`	*`COHERENCE_HOME`*`\lib\coherence-jpa.jar`	A 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. **Note:** The persistence unit is assumed to be set to use `RESOURCE_LOCAL` transactions.
`JpaCacheLoader`		A 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. Use the `JpaCacheStore` class for a full load and store implementation.
`EclipseLinkJPACacheStore`	*`ORACLE_HOME`*`\oracle_common\modules\oracle.toplink_`*`version`*`\toplink-grid.jar`	An EclipseLink specific JPA implementation of the Coherence `CacheStore` interface. This implementation is intended to be used where the application uses Coherence directly and the cache store and loader is used behind the scene to persist and load data. **Note:** To use this implementation, make sure no cache interceptors or query redirectors from the EclipseLink-Coherence integration are set within the persistence unit for the specific class.
`EclipseLinkJPACacheLoader`		An EclipseLink specific JPA implementation of the Coherence `CacheLoader` interface. **Note:** To use this implementation, make sure no cache interceptors or query redirectors from the EclipseLink-Coherence integration are set within the persistence unit for the specific class.

Obtaining a JPA Provider Implementation

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

Oracle recommends using EclipseLink JPA– the reference implementation for the JPA 2.0 specification and also the JPA provider used in Oracle TopLink. EclipseLink provides a high-performance JPA implementation with many advanced features for caching, threading, and overall performance.

The EclipseLink JAR files (`eclipselink.jar`) is included in the Coherence installation and can be found in the `ORACLE_HOME\oracle_common\modules\oracle.toplink_`*`version`* folder.

Configuring a Coherence JPA Cache Store

Using JPA with Coherence requires configuring persistence properties and defining a cache that uses the `JpaCacheStore` implementation.

This section includes the following topics:

- Mapping the Persistent Classes

Mapping 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.

Configuring 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>
```

Configuring a Coherence Cache for JPA

Create a `coherence-cache-config.xml` file to override the default Coherence settings and define a caching scheme. The caching scheme includes a `<cachestore-scheme>` element that lists the JPA implementation 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 cache named `Employee` that caches instances of the `Employee` class. The cache is configured to use the `JpaCacheStore` implementation. 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>
```

Configuring the Persistence Unit

When using a JPA cache store or loader implementation, 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.

When using a JPA cache store or loader implementation, L2 (shared) caching should be disabled in your persistence unit. See the documentation for your provider. In EclipseLink, 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"/>
```

2

Integrating with Oracle Coherence GoldenGate HotCache

Applications that use Coherence caches can leverage the Oracle Coherence GoldenGate HotCache (HotCache) integration to allow external changes to a database to be propagated to objects in Coherence caches.
A detailed description of Oracle GoldenGate is beyond the scope of this documentation. If you are new to GoldenGate, install the appropriate Oracle GoldenGate for your database environment. See the GoldenGate documentation library at Oracle GoldenGate for Big Data 21c.

In addition, see the following documents:

- Preparing the Database for Oracle GoldenGate in *Using Oracle GoldenGate with Oracle Database*.
- Installing Oracle GoldenGate Classic for Big Data in *Installing and Upgrading Oracle GoldenGate for Big Data*.
- Oracle GoldenGate Java Delivery in *Administering Oracle GoldenGate for Big Data*.
- Configuring Java Delivery documents in the Oracle GoldenGate for Big Data 21c documentation library.

> **Note:**
>
> To use HotCache, you must have licenses for Oracle GoldenGate and Coherence Grid Edition. A HotCache Extend Client can be used with Oracle GoldenGate for Big Data 12c/19c and run with Java 8. For a HotCache client running as a cluster member, the cluster member must run with Java 11. The minimum release of Oracle GoldenGate for Big Data that is certified to run with Java 11 is 21.4.0.0.0. See Release Notes for Oracle GoldenGate for Big Data. Oracle recommends that you use the latest available patch. Examples of configuring Oracle GoldenGate scripts and properties in this chapter refer to Oracle GoldenGate for Oracle Database and Oracle GoldenGate for Big Data 21c.

This chapter includes the following sections:

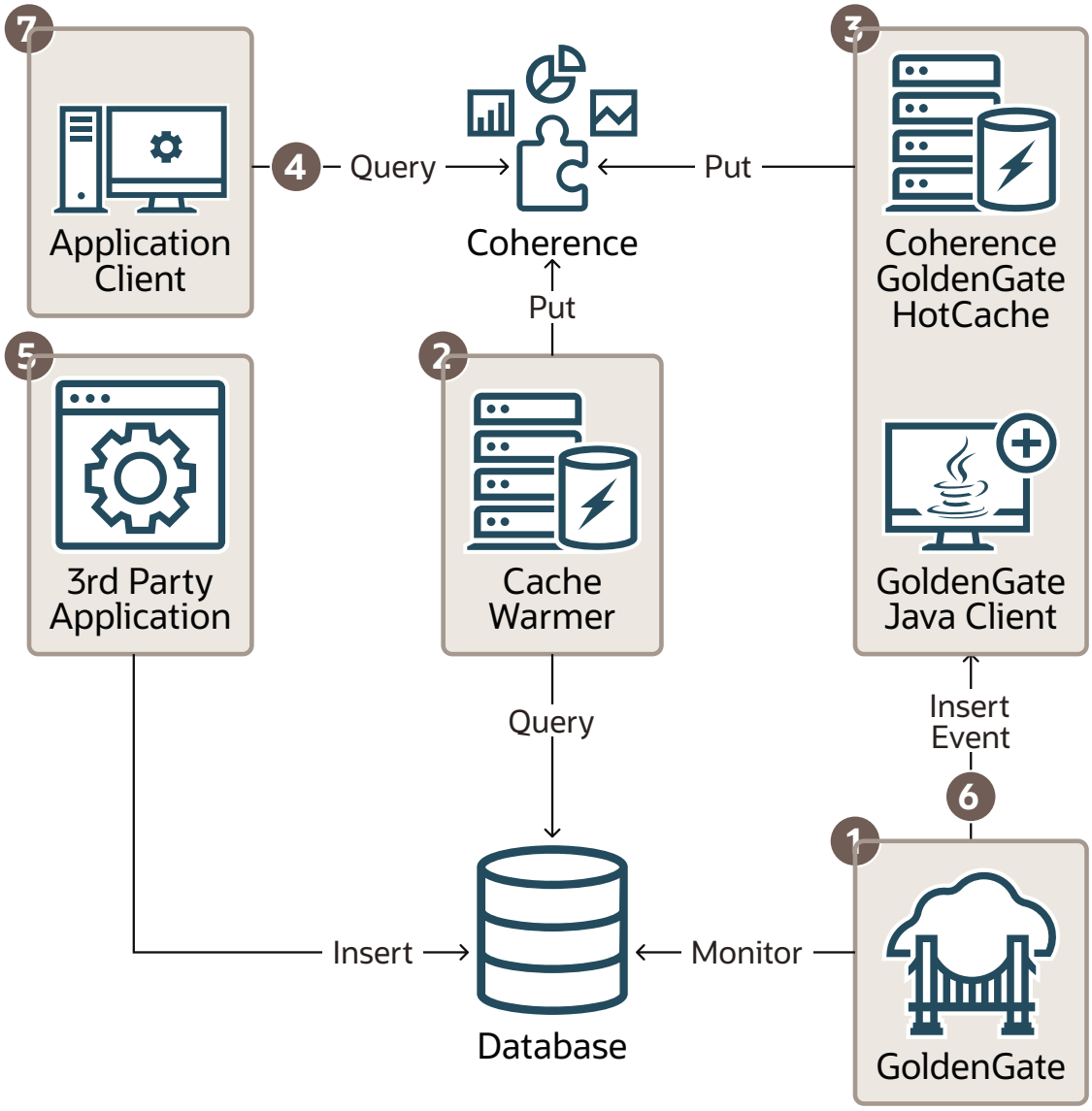
About Oracle Coherence GoldenGate HotCache

Third-party updates to the database can cause Coherence applications to work with data which could be stale and out-of-date. HotCache solves this problem by monitoring the database and pushing any changes into the Coherence cache. HotCache employs an efficient push model which processes only stale data. Low latency is assured because the data is pushed when the change occurs in the database.
HotCache can be added to any Coherence application. Standard JPA is used to capture the mappings from database data to Java objects. The configuration can be captured in XML exclusively or in XML with annotations.

The following scenario describes how HotCache could be used to work with the database and with applications that use Coherence caches. Figure 2-1 illustrates the scenario.

1. Start GoldenGate Extract, also referred to as Capture. GoldenGate monitors the transaction log for changes of interest. These changes will be placed into a "trail file". See GoldenGate Extract in the *Oracle GoldenGate Microservices Documentation*.
2. Start the Coherence cache server and warm the cache, if required.
3. Start HotCache so that it can propagate changes in the trail file into the cache. If changes occur during cache warming, then they will be applied to the cache once HotCache is started so no changes are lost.
4. Start an application client. As part of its operation, assume that the application performs repeated queries on the cache.
5. A third-party application performs a direct database update.
6. GoldenGate detects the database change which is then propagated to the Coherence cache by HotCache.
7. The application client detects the change in cache.

Figure 2-1 How HotCache Propagates Database Changes to the Cache



How Does HotCache Work

Before implementing a HotCache solution, take some time to understand HotCache fundamentals and supported features.
This section includes the following topics:

- Overview of How HotCache Works
- How the GoldenGate Java Delivery Adapter Uses JPA Mapping Metadata
- Supported Database Operations
- JPA Relationship Support

Overview of How HotCache Works

HotCache processes database change events delivered by GoldenGate and maps those changes onto the affected objects in the Coherence cache. It is able to do this through the use of Java Persistence API (JPA) mapping metadata. JPA is the Java standard for object-relational mapping in Java and it defines a set of annotations (and corresponding XML) that describe how Java objects are mapped to relational tables. As Example 2-1 illustrates, instances of an `Employee` class could be mapped to rows in an `EMPLOYEE` table with the following annotations.

Example 2-1 Mapping Instances of Employee Class to Rows with Java Code

```
@Entity
@Table(name="EMPLOYEE")
Public class Employee {
   @Id
   @Column(name="ID")
   private int id;
   @Column(name="FIRSTNAME")
   private String firstName;
…
}
```

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

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

Example 2-2 Simplified Java Code for Mapping Instances of Employee Class to Rows

```
@Entity
Public class Employee {
   @Id
   private int id;
   private String firstName;
…
}
```

Note that the Java code in the previous examples can also be expressed as XML. Example 2-3 illustrates the XML equivalent of the Java code in Example 2-1.

Example 2-3 Mapping Instances of Employee Class to Rows with XML

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

```
                ...
        </attributes>
</entity>
```

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

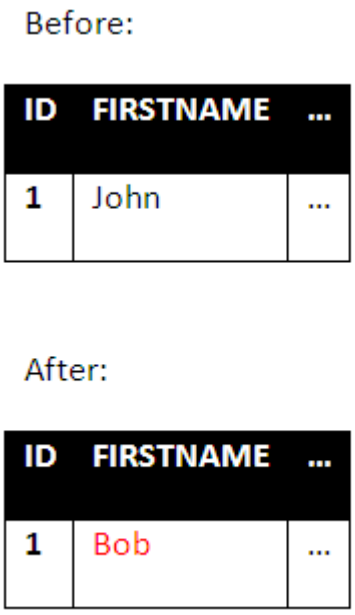
Example 2-4 Simplified XML for Mapping Instances of Employee Class to Rows

```
<entity class="Employee">
       <attributes>
            <id name="id"/>
            <basic name="firstName"/>
             ...
        </attributes>
</entity>
```

How the GoldenGate Java Delivery Adapter Uses JPA Mapping Metadata

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

Figure 2-2 EMPLOYEE Table Before and After an Update



With GoldenGate monitoring changes to the `EMPLOYEE` table and HotCache configured on the appropriate trail file, the adapter processes an event indicating the `FIRSTNAME` column of the `EMPLOYEE` row with primary key `1` has been changed to `Bob`. The adapter will use the JPA mapping metadata to first identify the class associated with the `EMPLOYEE` table, `Employee`, and then determine the column associated with an `Employee`'s ID field, `ID`. With this information, the

adapter can extract the ID column value from the change event and update the `firstName` field (associated with the `FIRSTNAME` column) of the `Employee` cached under the `ID` column value.

Supported Database Operations

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

JPA Relationship Support

HotCache does not support the JPA relationship mappings one-to-one, one-to-many, many-to-one, and many-to-many. However HotCache does support JPA embeddable classes and JPA element collections. Embeddable classes and element collections can be used with HotCache to model relationships between domain objects. Domain objects used with HotCache may also refer to each other by an identifier (analogous to foreign keys in a relational database).

As a performance optimization, when using JPA element collections with HotCache, it is suggested to configure GoldenGate with an `ADD TRANDATA` command specifying the column in the element collection table that is the foreign key to the parent table. The optimization allows HotCache to efficiently find the cache entry to update when a row in the element collection table changes.

Prerequisites

Ensure that you complete the prerequisites prior to using Oracle Coherence GoldenGate HotCache. The instructions assume that you have set up your database to work with GoldenGate.
Setting up a database includes:

- creating a database and tables
- granting user permissions
- enabling logging
- provisioning the tables with data

Example 2-5 illustrates a list of sample commands for the Oracle Database that creates a user named `csdemo` and grants user permissions to the database.

Note the `ALTER DATABASE ADD SUPPLEMENTAL LOG DATA` command. When supplemental logging is enabled, all columns are specified for extra logging. At the very least, minimal database-level supplemental logging must be enabled for any change data capture source database. If the values of primary key columns in a database table can change, it is important to include the following commands for Oracle Database: `ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;` and `ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (UNIQUE) COLUMNS;`.

Example 2-5 Sample Commands to Create a User, Grant Permissions, and Enable Logging

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

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

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

A detailed description of these tasks is beyond the scope of this documentation. See:

- Installing Oracle GoldenGate in the *Oracle GoldenGate Microservices Documentation*.
- Installing Oracle GoldenGate for Big Data in *Installing and Upgrading Oracle GoldenGate for Big Data*.
- Configuring Oracle GoldenGate for Oracle Database in the *Oracle GoldenGate Microservices Documentation*.
- Oracle GoldenGate Java Delivery in *Administering Oracle GoldenGate for Big Data*.

Configuring GoldenGate

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

> **Note:**
>
> The sample scripts provided in this section are intended only to be introductory. For a comprehensive list of configuration tasks, see Configuring Oracle GoldenGate for Oracle in the *Oracle GoldenGate Microservices Documentation* and Getting Started with Oracle GoldenGate (Classic) for Big Data in *Using Oracle GoldenGate for Big Data*.

This section includes the following topics:

Monitor Table Changes

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

```
ADD TRANDATA csdemo.*
```

Sample GoldenGate Capture ggsci Script to Monitor Table Changes illustrates a sample `ggsci` script named `cs-cap.ggsci`.

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

Example 2-6 Sample GoldenGate Capture `ggsci` Script to Monitor Table Changes

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

Filter Changes Made by the Current User

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

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

```
TranLogOptions excludeUser csdemo
```

Sample Extract .prm File for the GoldenGate Capture illustrates a sample extract `.prm` file named `cs-cap.prm`. The user that the Coherence CacheStore is logged in as is `csdemo`. The `EXTRAIL` parameter identifies the trail as `dirdat/cs`. The `BR BROFF` parameter controls the Bounded Recovery (BR) feature. The `BROFF` value turns off Bounded Recovery for the run and for recovery. The `GETUPDATEBEFORES` parameter indicates that the before images of updated columns are included in the records that are processed by Oracle GoldenGate. The `TABLE` parameter identifies `csdemo.*` as the tables that should be monitored for changes. The

`TranLogOptions excludeUSER` parameter indicates that GoldenGate should ignore changes to the tables made by the Coherence CacheStore user logged in as `csdemo`.

> **Note:**
>
> The `OverwriteMode` option is not applicable in Oracle GoldenGate for Big Data.

Example 2-7 Sample Extract `.prm` File for the GoldenGate Capture

```
EXTRACT cs-cap
USERID csdemo, PASSWORD csdemo
LOGALLSUPCOLS
UPDATERECORDFORMAT COMPACT
EXTTRAIL dirdat/cs
BR BROFF
getUpdateBefores
TABLE csdemo.*;
TranLogOptions excludeUser csdemo --ignore changes made by csuser
```

For details on available configuration options for capture, see Extract in the *Oracle GoldenGate Microservices Documentation*.

Configuring HotCache

HotCache is configured with system properties, EclipseLink JPA mapping metadata, and a JPA `persistence.xml` file. See How Does HotCache Work. The connection from HotCache to the Coherence cluster can be made by using Coherence*Extend (TCP), or the HotCache JVM can join the Coherence cluster as a member.
The following sections describe the properties needed to configure HotCache and provide details about connecting with Coherence*Extend:

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

Create a Properties File with GoldenGate for Java Properties

Create a text file with the filename extension `.properties`. In the file, enter the configuration for HotCache. A minimal configuration should contain the list of event handlers and the fully-qualified Java class of the event handler.

> **Note:**
>
> The path to the `.properties` file must be set in the HotCache replicat `TARGETDB` parameter in a `.prm` file, for example:
>
> ```
> TARGETDB LIBFILE libggjava.so SET property=/home/oracle/gg/hotcache.properties
> ```
>
> See Edit the HotCache Replicat Parameter File.

Example 2-8 illustrates a `.properties` file that contains the minimal configuration for a HotCache project. The following properties are used in the file:

- `gg.handlerlist=hotcache`

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

- `gg.handler.hotcache.type=[oracle.toplink.goldengate.CoherenceAdapter|oracle.toplink.goldengate.CoherenceAdapter1220]`

  The `gg.handler.{name}.type` property defines the handler for HotCache. The `{name}` field should be replaced with the name of an event handler listed in the `gg.handlerlist` property. The only handlers that can be set for HotCache are `oracle.toplink.goldengate.CoherenceAdapter` or `oracle.toplink.goldengate.CoherenceAdapter1220`. Use `oracle.toplink.goldengate.CoherenceAdapter1220` with GoldenGate Application Adapters release 12.2.0 or later. Use `oracle.toplink.goldengate.CoherenceAdapter` with GoldenGate Application Adapters releases earlier than 12.2.0.

- `gg.classpath` files

  The following is a list of directories and JAR files for the gg.handler(s) and their dependencies.

  - `coherence-hotcache.jar` – contains the Oracle Coherence GoldenGate HotCache libraries
  - `javax.persistence.jar` – contains the Java persistence libraries
  - `eclipselink.jar` – contains the EclipseLink libraries
  - `toplink-grid.jar` – contains the Oracle TopLink libraries required by HotCache
  - domain classes – the JAR file or directory containing the user classes cached in Coherence that are mapped with JPA for use in HotCache. Also, the Coherence configuration files, `persistence.xml` file, and any `orm.xml` file.

There are many other properties that can be used to control the behavior of the GoldenGate Java Delivery. See Java Delivery Properties in *Administering Oracle GoldenGate for Big Data*.

Example 2-8 .properties File for a HotCache Project

```
# ============================================================================
# List of active event handlers
# ============================================================================
gg.handlerlist=hotcache

# ============================================================================
# HotCache event handler
# ============================================================================
gg.handler.hotcache.type=oracle.toplink.goldengate.CoherenceAdapter1220

# ====================================
# HotCache handler dependency jars
# ====================================
# Set gg.classpath with following:
# persistence unit name, application jar(s), directory containing coherence
configuration files,
# $GGBD_HOME/dirprm, coherence.jar, coherence-hotcache, jar, eclipselink.jar,
# jakarta.persistence.jakarta.persistence-api.jar, and toplink-grid.jar from a
```

```
Coherence
# installation, as well as a JDBC driver jar for your database.
gg.classpath=<list of jars and directories separated by OS specific classpath separator>

# ====================================
# Options for HotCache JVM
# ====================================
jvm.bootoptions=-Djava.class.path=dirprm:ggjava/ggjava.jar -Xmx512M -Xms32M -
Dtoplink.goldengate.persistence-unit=employee -Dcoherence.distributed.localstorage=false
-Dcoherence.cacheconfig=/home/oracle/cgga/workspace/CacheStoreDemo/client-cache-
config.xml

# Note that if you are using a windows machine, you need to replace the : with a ; for
both gg.classpath and java.class.path.
```

Add JVM Boot Options to the Properties File

This section describes the properties that must appear in the JVM boot options section of the `.properties` file. These options are defined by using the `jvm.bootoptions` property. A sample `jvm.bootoptions` listing is illustrated in `JVM boot options` section of Example 2-8.

This section includes the following topics:

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

Java Classpath Files

The following is a list of directories and JAR files that should be included in the `java.class.path` property.

- `ggjava.jar` – contains the GoldenGate Java Delivery Adapter libraries
- `dirprm` – the GoldenGate `dirprm` directory

> **Note:**
>
> The `dirprm` directory is included here since it could include custom logging properties file required for logging initialization that occurs before `gg.classpath` is added to classloader. This directory can be moved to `gg.classpath` if it does not include any logging property or jar files. See Configuring Java Delivery.

HotCache-related Properties

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

The `toplink.goldengate.on-error` property is optional. It controls how the adapter responds to errors while processing a change event. This response applies to both expected optimistic lock exceptions and to unexpected exceptions. This property is optional, as its value defaults to

"Refresh". Refresh causes the adapter to attempt to read the latest data for a given row from the database and update the corresponding object in the cache. Refresh requires a database connection to be specified in the `persistence.xml` file. This connection will be established during initialization of HotCache. If a connection cannot be made, then an exception is thrown and HotCache will fail to start.

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

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

Coherence-related Properties

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

```
-Dcoherence.distributed.localstorage=false
```

Logging Properties

To configure Java Delivery logging for Oracle GoldenGate for Big Data, see Logging Properties in *Administering Oracle GoldenGate for Big Data* . In the Oracle GoldenGate for Big Data installation directory, examples of logging properties files are available for `jdk`, `logback`, and `log4j2` under the `AdapterExamples/java-delivery/sample-dirprm` directory.

Provide Coherence*Extend Connection Information

The connection between HotCache and the Coherence cluster can be made with Coherence*Extend. For more information on Coherence*Extend, see Developing Remote Clients for Oracle Coherence.

The Coherence configuration files must be in a directory referenced by the `gg.classpath` entry in the `.properties` file. For an example, see the `gg.classpath` files.

Example 2-9 illustrates the section of a client cache configuration file that uses Coherence*Extend to connect to the Coherence cluster. In the client cache configuration file, Coherence*Extend is configured in the `<remote-cache-scheme>` section. For additional options for configuring a remote-cache-scheme, see Overview of Configuring Extend Clients in *Developing Remote Clients for Oracle Coherence*.

Example 2-9 Coherence*Extend Section of a Client Cache Configuration File

```
<cache-config>
 ...
   <caching-schemes>
      <remote-cache-scheme>
         <scheme-name>CustomRemoteCacheScheme</scheme-name>
         <service-name>CustomExtendTcpCacheService</service-name>
         <initiator-config>
            <tcp-initiator>
               <remote-addresses>
```

```
                  <socket-address>
                     <address>localhost</address>
                     <port>9099</port>
                  </socket-address>
               </remote-addresses>
            </tcp-initiator>
            <outgoing-message-handler>
             ...
            </outgoing-message-handler>
         </initiator-config>
      </remote-cache-scheme>
 ...
</cache-config>
```

Example 2-10 illustrates the section of a server cache configuration file that listens for Coherence*Extend connections. In the server cache configuration file, Coherence*Extend is configured in the `<proxy-scheme>` section. By default, the listener port for Coherence*Extend is `9099`.

Example 2-10 Coherence*Extend Section of a Server Cache Configuration File

```
<cache-config>
   ...
   <caching-schemes>
    ...
      <proxy-scheme>
         <scheme-name>CustomProxyScheme</scheme-name>
         <service-name>CustomProxyService</service-name>
         <thread-count>2</thread-count>
         <acceptor-config>
            <tcp-acceptor>
               <local-address>
                  <address>localhost</address>
                  <port>9099</port>
               </local-address>
            </tcp-acceptor>
         </acceptor-config>
         <load-balancer>proxy</load-balancer>
         <autostart>true</autostart>
      </proxy-scheme>

   </caching-schemes>
</cache-config>
```

Configuring the GoldenGate Big Data Java Delivery Adapter

The GoldenGate Java Delivery Adapter provides a way to process GoldenGate data change events in Java by configuring an event handler class.The configuration for the GoldenGate Java Delivery Adapter allows it to monitor an a trail file and to pass data change events to HotCache. The configuration is provided in a replicat parameter and is described in this section.
This section includes the following topic:

- Edit the HotCache Replicat Parameter File

Edit the HotCache Replicat Parameter File

This section describes the parameters that can be defined in the replicat `.prm` file for a GoldenGate Big Data Java Delivery adapter. The parameters that are illustrated in Example 2-11 constitute a minimal configuration for a HotCache project.

For details on creating a replicat parameter file, see Basic Parameters for Different Replicat Modes in the *Oracle GoldenGate Microservices Documentation*.

- `TARGETDB LIBFILE libggjava.so SET property=/home/oracle/gg/hotcache.properties`
- `GROUPTRANSOPS 1`

  The `GROUPTRANSOPS` parameter controls transaction grouping by the GoldenGate replicat process. A value of `1` tells the GoldenGate replicat process to honor source database transaction boundaries in the trail file. A value greater than `1` tells the GoldenGate replicat process to group operations from multiple source database transactions into a single target transaction. See GROUPTRANSOPS in *Reference for Oracle GoldenGate for Windows and UNIX*.

- `MAP scott.*, TARGET scott.*;`

  The MAP parameter tells the GoldenGate replicat process how to map source database tables to the replication target. The parameter syntax assumes the replication target is a relational database. For HotCache it is appropriate to specify an identical mapping. See TABLE and MAP Options in *Reference for Oracle GoldenGate*.

Sample .prm Parameter File for a GoldenGate Big Data Java Delivery adapter illustrates a sample `.prm` file for a GoldenGate Big Data Java Delivery adapter.

Example 2-11 Sample .prm Parameter File for a GoldenGate Big Data Java Delivery adapter

```
REPLICAT hotcache
TARGETDB LIBFILE libggjava.so SET property=/home/user/project/hotcache.properties
GROUPTRANSOPS 1
GetUpdateBefores
MAP scott.*, TARGET scott.*;
```

Configuring the Coherence Cache Servers

You must modify the classpaths of all Coherence cache server JVMs that contain caches that are refreshed by HotCache. Place the following JAR files, included in the Coherence installation, on each cache server classpath:

- `coherence-hotcache.jar` – contains the Oracle Coherence GoldenGate HotCache libraries
- `javax.persistence.jar` – contains the Java persistence libraries
- `eclipselink.jar` – contains the EclipseLink libraries
- `toplink-grid.jar` – contains the Oracle TopLink libraries required by HotCache
- domain classes – the JAR file or directory containing the user classes cached in Coherence that are mapped with JPA for use in HotCache.

Using Portable Object Format with HotCache

Serialization is the process of encoding an object into a binary format. It is a critical component to working with Coherence as data must be moved around the network. Portable Object Format (also known as POF) is a language-agnostic binary format. POF was designed to be very efficient in both space and time and has become a cornerstone element in working with Coherence. POF serialization can be used with HotCache but requires a small update to the POF configuration file (`pof-config.xml`) to allow for HotCache and TopLink Grid framework classes to be registered.
The `pof-config.xml` file must include the `coherence-hotcache-pof-config.xml` file and must register the `TopLinkGridPortableObject` user type and `TopLinkGridSerializer` as the serializer. The `<type-id>` for each class must be unique and must match across all cluster instances. See Registering POF Objects in *Developing Applications with Oracle Coherence*.

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

Example 2-12 illustrates a sample `pof-config.xml` file for HotCache. The value *`integer_value`* represents a unique integer value greater than `1000`.

Example 2-12 Sample POF Configuration File for HotCache

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

Configuring HotCache JPA Properties

You can customize HotCache using a number of custom JPA properties that can be configured per JPA entity type. These properties can be configured either by an `@Property` annotation on the JPA entity class or by a `<property>` element in the `persistence.xml` file. The latter takes precedence in the event of conflicting configuration.
This section includes the following topics:

- EnableUpsert Property

EnableUpsert Property

EnableUpsert

The `EnableUpsert` property controls whether HotCache inserts a cache entry when an update operation is received in the GoldenGate trail file but no corresponding cache entry is present in cache at the entity key. By default, HotCache ignores updates to absent entities. Set this property to `true` if you want HotCache to insert missing entities into the cache when update operations are received in the trail file. The default value of this property is `false`.

Setting this property to `true` can facilitate warming caches in an event-driven manner if it is likely that entities will be accessed from the cache after their corresponding records are updated in the underlying database.

> **Note:**
>
> There are risks to consider when using this property:
>
> - The entity to insert is read from the database, as the trail file may not contain values for all fields of the entity to be inserted. This can reduce the throughput of the HotCache process by waiting on database reads.
> - Cache capacity can be exhausted if more rows in the DB are updated than the number of entities in the cache for which capacity was provisioned.

Entity Class Annotation

```
@Property(name = "EnableUpsert", value = "false",valueType = boolean.class)
```

Persistence XML Property

```
<property name="[fully qualified entity class name].EnableUpsert" value="[true|false]"/>
```

HonorRedundantInsert Property

HonorRedundantInsert

The `HonorRedundantInsert` property controls whether HotCache honors an insert operation in the GoldenGate trail file when a cache entry at that key is already present. By default, HotCache ignores a redundant insert operation. However, when a JPA entity is mapped to a complex materialized view in Oracle Database and a row is updated in a table underlying that materialized view (thus updating one or more rows of the materialized view), Oracle Database inserts a new row into the materialized view with the same PK as an existing row but with a new rowid and deletes the existing row. Therefore, HotCache sees a redundant insert operation that really represents an update to the cached JPA entity. Users in this situation should also consider suppressing replication of delete operations on that materialized view through the use of GoldenGate configuration; otherwise, the cached entity is deleted by HotCache. The default value of this property is `false`.

Entity Class Annotation

```
@Property(name = "HonorRedundantInsert", value = "false",valueType = boolean.class)
```

Persistence XML Property

```
<property name="[fully qualified entity class name].HonorRedundantInsert" value="[true|
false]"/>
```

SyntheticEvent Property

SyntheticEvent

The `SyntheticEvent` property controls whether cache writes by HotCache are synthetic or not. Synthetic writes to Coherence caches do not trigger events in Coherence; they do not engage Federated Caching; and they do not call Coherence `CacheStore` implementations. Set this property to `false` for a JPA entity class if you want cache writes by HotCache for that class to be non-synthetic so that events are triggered, Federated Caching is engaged, and `CacheStore` implementations are called (if any of those are configured for the entity class cache). The default value of this property is `true` for every entity class.

> **Note:**
>
> There is a risk of infinite replication loops if the `SyntheticEvent` is set to `true` for an entity class and a `CacheStore` implementation is configured on that entity class cache and writing to the same database HotCache is replicating to Coherence. This risk can be mitigated by filtering transactions by database user. See Filter Changes Made by the Current User.

Entity Class Annotation

```
@Property(name = "SyntheticEvent", value = "[true|false]", valueType = boolean.class)
```

Persistence XML Property

```
<property name="[fully qualified entity class name].SyntheticEvent" value="[true|
false]"/>
```

eclipselink.cache.shared.default Property

eclipselink.cache.shared.default

The `eclipselink.cache.shared.default` property is used to enable the EclipseLink internal shared cache. It is important to disable the internal shared cache in the HotCache JVM by setting the property to `false` in the `persistence.xml` file.

Persistence XML Property

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

Warming Caches with HotCache

HotCache can be used to warm caches by loading an initial dataset. This approach eliminates the need to write custom cache warming programs because it leverages GoldenGate and HotCache for initial cache warming.
This section includes the following topics:

- Create and Run an Initial Load Extract
- Create and Run a Cache Warmer Replicat
- Capturing Changed Data While Warming Caches

Create and Run an Initial Load Extract

To create and run and initial load extract:

1. Create a GoldenGate extract parameter file named `initload.prm` as shown below and save it to *GG_HOME*`/dirprm`. Note that the extract files cannot have filenames longer than eight characters. A GoldenGate extract process that is run with this parameter file selects records from the source database (as opposed to capturing changes from the database's transaction log) and writes them to a trail file in canonical format.

   ```
   -- This is an initial load extract initload
   -- SOURCEISTABLE parameter indicates source is a table, not redo logs
   SOURCEISTABLE
   USERID <user>, PASSWORD <password>
   -- EXTFILE parameter indicates path and prefix of data files
   -- Note: set MEGABYTES parameter to a maximum file size relative
   -- to the amount of source data being extracted
   EXTFILE GG_HOME/dirdat/IL, maxfiles 9999, MEGABYTES 5, PURGE
   TABLE <schema>.*
   ```

2. Using the above extract parameters file, run a GoldenGate initial load extract process directly from the command line as shown in the following example.

   ```
   cd GG_HOME
   extract paramfile GG_HOME/dirprm/initload.prm reportfile GG_HOME/dirrpt/initload.rpt
   ```

After running the extract process, there will be one or more trail files named `IL0001`, `IL0002`, etc... in the *GG_HOME*`/dirdat` directory. If no files are generated, then review the *GG_HOME*`/dirrpt/initload.rpt` file.

Create and Run a Cache Warmer Replicat

To create and run a cache warmer replicat:

1. Create a GoldenGate replicat parameter file named `warmcach.prm` as shown in the example below. A GoldenGate replicat process that is run with this parameter file reads the initial load dataset from the trail files. See Create and Run an Initial Load Extract.

   ```
   REPLICAT warmcach
   TARGETDB LIBFILE libggjava.so SET property=/home/user/project/
   warmcach.properties
   MAP <schema>.*, TABLE <schema>.*
   ```

2. Since the replicat parameter file uses the GoldenGate Java Delivery Adapter, create a corresponding `warmcach.properties` file in *GGBD_HOME*`/dirprm` as shown in the example below.

```
#=============================================================
# List of active event handlers
gg.handlerlist=hotcache
#=============================================================
# HotCache handler
gg.handler.hotcache.type=oracle.toplink.goldengate.CoherenceAdapter1220
#=============================================================
# Options for the HotCache gg.classpath.
gg.classpath=
# Obviously the persistence unit name, classpath, and other
# options will vary between users and environments. The gg.classpath
# must include $GGBD_HOME/dirprm,
# coherence.jar, coherence-hotcache, jar, eclipselink.jar,
# jakarta.persistence.jakarta.persistence-api.jar, and toplink-grid.jar from a
Coherence
# installation, as well as a JDBC driver jar for your database,
# and a jar with your cache key and value classes in it.
#=============================================================
# Options for the HotCache JVM
# Other system properties may override Coherence operational
# configuration elements for cluster addresses and names,
# paths to configuration files, etc. You may also wish to provide
# non-default JVM heap sizes, logging configuration, etc.
jvm.bootoptions=-Djava.class.path=dirprm:ggjava/ggjava.jar -Xmx512M -Xms32M -
Dtoplink.goldengate.persistence-unit=pu_name -
Dcoherence.distributed.localstorage=false
```

3. Register the `warmcach` replicat process with the GoldenGate installation using the GoldenGate GGSCI command-line interface as shown in the following example.

```
cd $GGBD_HOME
./ggsci
add replicat warmcach, exttrail GGBD_HOME/dirrpt/IL
```

   The replicat parameter file and properties file above are used by the GoldenGate `warmcach` process that reads the trail files created by the initial load extract process.

4. Run the GoldenGate `warmcach` replicat process by issuing the following commands through the GoldenGate GGSCI command-line interface.

```
cd $GGBD_HOME
./ggsci
start mgr
start replicat warmcach
```

   After the `warmcach` replicat process has finished running, the contents of the initial load trail files will have been transformed into JPA entities and put into Coherence caches.

5. Stop and unregister the `warmcach` replicat, using the following GGSCI commands.

```
stop replicat warmcach
delete replicat warmcach
```

Capturing Changed Data While Warming Caches

HotCache was developed to refresh Coherence caches as underlying database transactions occur. Using HotCache for initial cache warming is an added benefit. It is possible to capture

changed data in the database while initial cache warming takes place and refresh Coherence caches with that changed data by following a carefully sequenced procedure. The necessary sequence of operations is as follows:

1. Start the normal source extract process that captures change data from the database's redo logs, but **do not** start the normal HotCache replicat process that refreshes Coherence caches with that change data.
2. Start the initial load extract process to select the initial data set from the database.
3. Run the cache warming replicat process to warm Coherence caches with the initial data set.
4. Verify that the initial load has completed correctly by comparing the number of rows extracted from the database by GoldenGate (see `initload.rpt`) with the number of entries in the target Coherence caches according to Coherence MBeans or command-line interface commands.
5. Start the normal HotCache replicat process to refresh Coherence caches with change data.

Implementing High Availability for HotCache

HotCache is a client of Coherence cache services and invokes the services to insert, update, or evict cache entries in response to transactions in an underlying database. As a cache client, HotCache can be configured either as a Coherence cluster member or as a Coherence*Extend client connecting to a Coherence proxy service in the cluster.
In best-practice deployments, Coherence cache services and proxy services are already highly available due to redundancy of service members (for example, multiple cache server processes and multiple proxy server processes) and due to built-in automatic failover capabilities within Coherence. For example, if a proxy server should fail, then Coherence*Extend clients that are using the proxy server automatically fail over to another proxy server. Likewise, if a cache server should fail then another cache server assumes responsibility for its data and client interactions with that data automatically redirect to the new cache server owning the data.

Making the HotCache client itself highly available relies on standard GoldenGate HA techniques since the HotCache JVM runs embedded in a GoldenGate process.

GoldenGate implements "single server" HA through `AUTOSTART` and `AUTORESTART` parameters enforced by the Manager process in a GoldenGate installation. The Manager process automatically starts registered GoldenGate processes configured with `AUTOSTART`. It also detects the death of (and automatically restarts), registered GoldenGate processes configured with `AUTORESTART`.

To protect against failure of the Manager process itself or the host on which it runs or the network connecting that host, GoldenGate relies on Oracle Clusterware to detect the death of the active GoldenGate installation and fail over to a passive GoldenGate installation.

GoldenGate high availability is discussed in the Oracle GoldenGate Classic Architecture with Oracle Real Application Clusters Configuration Best Practices white paper.

Support for Oracle Data Types

HotCache uses EclipseLink as its JPA provider. It is reasonable to expect HotCache to support Oracle-specific data types supported by EclipseLink. For example, EclipseLink supports data types specific to Oracle Database, such as `SDO_GEOMETRY` from the Oracle Spatial and Graph option for Oracle Database and `XMLType` in all Oracle Database editions.

It is important to understand that data is presented to EclipseLink differently when used in HotCache than when used in the typical JPA scenario. In the typical JPA scenario, EclipseLink interacts with the database through a JDBC connection and EclipseLink consumes data as presented by the JDBC API and driver-specific extensions (for example an `SDO_GEOMETRY` column is represented as an instance of `java.sql.Struct`). Whereas in HotCache, data is read from a GoldenGate trail file; there is no JDBC connection involved. Therefore EclipseLink consumes the GoldenGate representation of data as opposed to the JDBC representation of data. For example, GoldenGate represents an `SDO_GEOMETRY` column as an XML document and not as an instance of `java.sql.Struct`.

These differences in data representation may necessitate the use of HotCache-specific EclipseLink converters when using EclipseLink within HotCache that take the place of standard EclipseLink converters used in typical JPA scenarios. See @Converter in *Java Persistence API (JPA) Extensions Reference for EclipseLink*. The following sections describe HotCache support for specific Oracle Database data types supported by EclipseLink and how to configure EclipseLink within HotCache to use those data types.

- Support for SDO_GEOMETRY
- Support for XMLType

Support for SDO_GEOMETRY

EclipseLink supports the Oracle Spatial and Graph option of Oracle Database by mapping `SDO_GEOMETRY` columns to instances of `oracle.spatial.geometry.JGeometry` (the Java class shipped with the Oracle Spatial and Graph option). See Using Oracle Spatial and Graph in *Solutions Guide for EclipseLink*.

Therefore, HotCache supports mapping columns of type `SDO_GEOMETRY` to instances of `oracle.spatial.geometry.JGeometry` bound to fields of JPA entities. This support requires configuring a HotCache-specific EclipseLink Converter of class `oracle.toplink.goldengate.spatial.GoldenGateJGeometryConverter` as shown in the following example.

```
import javax.persistence.Access;
import javax.persistence.AccessType;
import javax.persistence.Convert;
import javax.persistence.Converter;
import javax.persistence.Entity;

import oracle.spatial.geometry.JGeometry;

import oracle.toplink.goldengate.spatial.GoldenGateJGeometryConverter;


@Entity
@Converter(name=”JGeometry”, converterClass=
GoldenGateJGeometryConverter.class)
public class SpatialEntity {

    private JGeometry geometry;

    @Access(AccessType.PROPERTY)
    @Convert(“JGeometry”)
    public JGeometry getGeometry() {
```

```
        return geometry;
    }
```

This converter converts the GoldenGate XML representation of an `SDO_GEOMETRY` column into an instance of `oracle.spatial.geometry.JGeometry` bound to a field of a JPA entity. The `GoldenGateJGeometryConverter` class is contained in `coherence-hotcache.jar` which should already be on the classpath of the HotCache JVM and Coherence cache server JVMs used in HotCache deployments (along with the `eclipselink.jar` file on which it depends). However the `JGeometry` class is contained in `sdoapi.jar` from an installation of Oracle Spatial and Graph option. The `sdoapi.jar` file must be on the classpath of the HotCache JVM, and any other JVM where the JPA entity containing a `JGeometry` field will be deserialized.

The `oracle.spatial.geometry.JGeometry` class implements `java.io.Serializable`, so JPA entities with `JGeometry` fields cached in Coherence can be serialized with `java.io.Serializable` without any additional configuration. To use Coherence's Portable Object Format (POF) to serialize a JPA entity with a `JGeometry` field, the `JGeometrySerializer` must be added to the POF configuration file used in the Coherence deployment, as in the following example.

```
<user-type>
   <type-id>1001</type-id><!—use a type-id value above 1000 that doesn't
conflict with other POF type-ids-->
   <class-name>oracle.spatial.geometry.JGeometry</class-name>
   <serializer>
       <class-name>oracle.spatial.geometry.JGeometryPofSerializer</class-name>
   </serializer>
</user-type>
```

The `oracle.spatial.geometry.JGeometryPofSerializer` class is contained in `coherence-hotcache.jar`, which must be on the classpath of any JVM that will serialize or deserialize a JPA entity with a `JGeometry` field using POF.

Support for XMLType

EclipseLink supports the Oracle Database `XMLType` data type by mapping `XMLType` columns to instances of `java.lang.String` or `org.w3c.dom.Document` (depending on the type of the mapped field in the JPA entity). See DirectToXMLTypeMapping in *EclipseLink API Reference* and Mapping XMLTYPE in the *On Persistence* blog.

Therefore, HotCache supports mapping columns of type `XMLType` to instances of `java.lang.String` or `org.w3c.dom.Document` bound to fields of JPA entities. This support requires configuring a standard EclipseLink `DirectToXMLTypeMapping` .

GoldenGate must be configured to use integrated capture mode for support of `XMLType` columns. See Details of Support for Oracle Data Types and Objects in *Using Oracle GoldenGate with Oracle Database*.

Configuring Multi-Threading in HotCache

HotCache can use multiple threads to apply trail file operations to Coherence caches. Multiple threads can increase the throughput of a HotCache process as compared to using a single thread to apply trail file operations. Before configuring multi-threading, evaluate whether

concurrently applying trail file operations poses data correctness risks in the Coherence caches and the system using HotCache.
Transactions and their operations appear in the trail file in the order in which they were committed in the source database. By default, HotCache applies operations one at a time on a single thread to ensure the operations are applied to the cache in the exact same order in which they were applied to the source database. When using multi-threading, operations can be applied in a different order than that in which they were applied to the source database tables and can result in correctness risks.

When determining the potential risk, consider the following examples:

- If one database transaction inserts a row in a table and the next database transaction deletes that row, then applying operations out of order can leave an object in the cache whose corresponding database row is deleted.
- If one database transaction updates a column to an older value and the next database transaction updates that column to a newer value, then applying operations out of order can leave the older value in the cached object instead of the newer value. (You can use the JPA optimistic locking features, which are supported by HotCache, to mitigate this particular update risk).

If you determine that using multiple threads to apply trail file operations to Coherence caches poses no data correctness risks in the system using HotCache, then HotCache can be configured to use multi-threading as follows:

1. Edit the GoldenGate Java Delivery Adapter properties file and configure the HotCache event handler to use transaction mode:

   ```
   gg.handlerlist=hotcache
   goldengate.handler.hotcache.type=oracle.toplink.goldengate.CoherenceAdapter
   1220
   goldengate.handler.hotcache.mode=tx
   ```

   By default, GoldenGate Java Delivery Adapter event handlers use operation mode. In operation mode (`op`), event handlers process operations one at a time. In transaction mode (`tx`), event handlers process all operations in a transaction at a time.

2. Edit the GoldenGate Java Delivery Adapter properties file and set the `coherence.hotcache.concurrency` system property on the HotCache JVM with a value between one and eight times the number of cores on the JVM host, inclusive (as reported by `java.lang.Runtime.getAvailableProcessors()`). For example:

   ```
   jvm.bootoptions=-Dcoherence.hotcache.concurrency=16 …
   ```

   The value of this property determines the number of threads HotCache uses to concurrently apply trail file operations to Coherence caches.

3. Edit the HotCache replicat `.prm` file and set the `GROUPTRANOPS` property. A value of `1` causes source database transaction boundaries to be honored. A value greater than `1` causes transaction grouping within the GoldenGate replicat. The default value is `1000`.

Summary of Hot Cache Thread Behavior

Assuming HotCache is run in a GoldenGate replicat process as recommended, the risks stemming from conflicting source database transactions only materialize if the `GROUPTRANOPS` property is configured to a value other than one `1`. A value of `1` causes the source database transaction boundaries and sequencing to be honored by the HotCache replicat. Therefore, the operations in one transaction are applied in parallel followed by the operations in the next

transaction and so on. The GROUPTRANOPS property default is 1000, which groups trail file operations from multiple successive source database transactions into one target transaction of at least 1,000 operations. The likelihood of data correctness risks materializing when the GROUPTRANOPS parameter is set to greater than one is equivalent to the likelihood of conflicting operations within the grouped source database transactions, given the magnitude of the GROUPTRANOPS property value and the write rate and volume of the source database. See GROUPTRANSOPS in *Reference for Oracle GoldenGate for Windows and UNIX*.

The following table summarizes the HotCache thread behavior depending on the values of the GoldenGate Java Delivery Adapter mode property and the coherence.hotcache.concurrency property.

Table 2-1 Hot Cache Thread Behavior

Mode	Concurrency	Behavior
op	N/A	In op mode, HotCache applies trail file operations one at a time on a single thread (the GoldenGate Java Delivery Adapter thread) as each operation is read from the trail file. This is the HotCache default behavior. The value of the concurrency property is not considered in operation mode.
tx	1	In tx mode with a concurrency property value of 1, HotCache iterates and applies a transaction worth of trail file operations on a single thread (the GoldenGate Java Delivery Adapter thread). The group of operations comprising the transaction is determined by the value of the GoldenGate replicat GROUPTRANOPS property. The default value of the concurrency property is 1. This configuration may exhibit greater throughput than the operation mode configuration, even though it is still single-threaded and therefore poses no data correctness risks.
tx	>1	In tx mode with a concurrency property value greater than 1, HotCache applies a transaction worth of operations in parallel on multiple HotCache threads. The group of operations comprising the transaction is determined by the value of the GoldenGate replicat GROUPTRANOPS property. This configuration should exhibit greater throughput than single-threaded configurations and throughput generally increases with the number of threads configured to a maximum of eight times the number of cores on the HotCache host.

Managing HotCache

You can manage HotCache to ensure that cache update operations are performed within acceptable time limits. HotCache uses JMX to collect management data, which is viewed using either a JMX browser, a Coherence report, or the Coherence-Java VisualVM plug-in. Management data includes statistics for the GoldenGate HotCache adapter as a whole in addition to statistics for specific caches and operation types.
This section includes the following topics:

- CoherenceAdapterMXBean
- Understanding the HotCache Report
- Monitoring HotCache Using the Coherence VisualVM Plug-In

CoherenceAdapterMXBean

The `CoherenceAdapterMXBean` MBean represents a Golden Gate HotCache adapter and provides operational and performance statistics. Zero or more instances of this managed bean are created: one managed bean instance for each adapter instance.

The object name of the MBean is:

`Type=CoherenceAdapter,name=`*`replicat name`*`,member=`*`member name`*

To view the `CoherenceAdapterMXBean` MBean from an MBean browser, you must enable Coherence management. If you are new to Coherence JMX management, see Using JMX to Manage Oracle Coherence.

Attributes

Table 2-2 describes the attributes for `CacheMBean`.

Table 2-2 CoherenceAdapterMXBean

Attribute	Type	Access	Description
`CacheNames`	`String[]`	read-only	The names of the caches that were refreshed by the CoherenceAdapter
`ExecutionTimePerOperationStatistics`	`LongSummaryStatistics`	read-only	Summary statistics about the execution time for each operation in nanoseconds since the statistics were last reset
`ExecutionTimePerTransactionStatistics`	`LongSummaryStatistics`	read-only	Summary statistics about the execution time for each transaction in nanoseconds since the statistics were last reset
`InvocationsPerOperationStatistics`	`IntSummaryStatistics`	read-only	Summary statistics about the number of invocations for each operation since the statistics were last reset
`LastExecutionTimePerOperationStatistics`	`LongSummaryStatistics`	read-only	Summary statistics about the execution time for each operation in nanoseconds since this method was last called
`LastOperationReplicationLagStatistics`	`LongSummaryStatistics`	read-only	Summary statistics about operation replication lag in milliseconds since this method was last called
`NumberOfOperationsProcessed`	`Long`	read-only	The aggregate number of operations processed since the statistics were last reset
`OperationReplicationLagStatistics`	`LongSummaryStatistics`	read-only	Summary statistics about operation replication lag in milliseconds since the statistics were last reset
`OperationsPerTransactionStatistics`	`IntSummaryStatistics`	read-only	Summary statistics about the number of operations for each transaction since the statistics were last reset
`PerCacheStatistics`	`Map`	read-only	Execution time summary statistics in nanoseconds for each cache for each operation type
`StartTime`	`Date`	read-only	The time at which the CoherenceAdapter was started

Table 2-2 (Cont.) CoherenceAdapterMXBean

Attribute	Type	Access	Description
`TrailFileName`	`String`	read-only	The name of the trail file currently being read
`TrailFilePosition`	`String`	read-only	The position in the trail file of the last successfully-processed operation

Operations

The `CoherenceAdapterMXBean` MBean includes a `resetStatistics` operation that resets all cache statistics.

Understanding the HotCache Report

The HotCache report includes operational settings and performance statistics. The statistical data is collected from the `CoherenceAdapterMXBean` MBean and presented over time making it ideal for discovering performance trends and troubleshooting potential performance issues. The name of the HotCache report is *`timestamp`*`-hotcache.txt` where the timestamp is in YYYYMMDDHH format. For example, a file named `2009013101-hotcache.txt` represents a HotCache report for January 31, 2009 at 1:00 a.m.

To view the HotCache report, you must enable Coherence reporting and you must configure the report-all report group. If you are new to Coherence reporting, see Using Oracle Coherence Reporting.

Table 2-3 describes the contents of the HotCache report.

Table 2-3 Contents of the HotCache Report

Column	Data Type	Description
`Batch Counter`	`Long`	A sequential counter to help integrate information between related files. This value resets when the reporter restarts, and is not consistent across members. However, it is helpful when trying to integrate files.
`Report Time`	`Date`	A timestamp for each report refresh
`Handler Name`	`String`	The user-given name of the HotCache event handler from the GoldenGate HotCache properties file
`Member Name`	`String`	The Coherence member name where the HotCache adapter runs
`Start Time`	`Date`	The time when the Coherence HotCache adapter started
`Operations Processed`	`Long`	The number of transaction operations processed
`Trail File Name`	`String`	The name of the Golden Gate trail file that contains transaction operations
`Trail File Position`	`String`	The position in the trail file of the last successfully-processed operation
`Operations per Transaction Average`	`IntSummaryStatistics`	The average number of operations processed for each transaction

Table 2-3 (Cont.) Contents of the HotCache Report

Column	Data Type	Description
Operations per Transaction Maximum	IntSummaryStatistics	The maximum number of operations processed for each transaction
Operations per Transaction Minimum	IntSummaryStatistics	The minimum number of operations processed for each transaction
Invocations per Operation Average	IntSummaryStatistics	The average number of entry processor invocations that are performed for each operation
Invocations per Operation Maximum	IntSummaryStatistics	The maximum number of entry processor invocations that are performed for each operation
Invocations per Operation Minimum	IntSummaryStatistics	The minimum number of entry processor invocations that are performed for each operation
Last Execution Time per Operation Average (ns)	LongSummaryStatistics	The average execution time for each operation since the last sample in nanoseconds
Execution Time per Operation Average (ns)	LongSummaryStatistics	The average execution time for each operation in nanoseconds
Execution Time per Operation Maximum (ns)	LongSummaryStatistics	The maximum execution time for each operation in nanoseconds
Execution Time per Operation Minimum (ns)	LongSummaryStatistics	The minimum execution time for each operation in nanoseconds
Execution Time per Transaction Average (ns)	LongSummaryStatistics	The average execution time for each transaction in nanoseconds
Execution Time per Transaction Maximum (ns)	LongSummaryStatistics	The maximum execution time for each transaction in nanoseconds
Execution Time per Transaction Minimum (ns)	LongSummaryStatistics	The maximum execution time for each transaction in nanoseconds
Last Operation Replication Lag Average (ms)	LongSummaryStatistics	The average time in milliseconds between the commit of the database transaction and the processing of the last operation by the HotCache adapter
Operation Replication Lag Average (ms)	LongSummaryStatistics	The average time in milliseconds between the commit of the database transaction and the processing of the operation by the HotCache adapter
Operation Replication Lag Maximum (ms)	LongSummaryStatistics	The average time in milliseconds between the commit of the database transaction and the processing of the operation by the HotCache adapter since the last sample
Operation Replication Lag Minimum (ms)	LongSummaryStatistics	The minimum time in milliseconds between the commit of the database transaction and the processing of the operation by the HotCache adapter

Monitoring HotCache Using the Coherence VisualVM Plug-In

The HotCache tab in the Coherence VisualVM Plug-In provides a graphical view of HotCache performance statistics. If you are new to the Coherence VisualVM plug-in, see Using the Coherence VisualVM Plug-In.

The HotCache statistical data is collected from the `CoherenceAdapter` MBean and presented over time in both tabular and graph form. The tab displays statistics for each GoldenGate HotCache member including detail about specific caches refreshed by that HotCache member. To view data for a specific member, select the member on the member table. To view data for a specific cache, select the cache on the cache table.

Use the HotCache tab to get a detailed view of performance statistics and to identify potential performance issues with cache update operations. The HotCache tab includes:

- The minimum, maximum, and average time it takes to update a cache for each operation.
- The minimum, maximum, and average time it takes to update a cache for all the operations in a transaction.
- The total number of entry processor invocations that are performed for each operation.
- The minimum, maximum, and average time for the last operation.
- The minimum, maximum, and average operation replication lag time for the last operation since this MBean attribute value was last sampled. Replication lag is the amount of time between the commit of the database transaction and the processing of the operation by the HotCache adapter.
- The minimum, maximum, and average operation replication lag time since the statistics were last reset.
- The minimum, maximum, and average number of operations for each transaction.
- The minimum, maximum, and average time for each operation type for each cache. Operations include: `EVICT`, `INSERT`, `PK_CHANGE`, `READ_FROM_DB`, `REDUNDANT_INSERT`, `REFRESH`, `UPDATE`, and `UPSERT`.

Fixing Issues

If you run into issues while using HotCache with Coherence, you may be able to resolve the issue using one of the following options.

Issue: The following error appears:

```
WARNING: Illegal reflective access by
org.eclipse.persistence.internal.security.PrivilegedAccessHelper (org/eclipse/
persistence/eclipselink/<version>/eclipselink-<version>.jar) to method
java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int)
```

Fix: Add the following JPMS option to the Java command line: `--add-opens java.base/java.lang=org.eclipse.persistence.core`

For information on why this JPMS option is necessary, see Using Java Modules to Build a Coherence Application in *Developing Applications with Oracle Coherence*.

3
Integrating Hibernate and Coherence

Oracle Coherence can be integrated with Hibernate, an object-relational mapping tool for Java environments. The functionality in Oracle Coherence and Hibernate can be combined such that Hibernate can act as the Coherence cache store or Coherence can act as the Hibernate L2 cache.

If you are interested in using Coherence with Hibernate, see the Coherence Hibernate Integration project that is part of the Coherence Community. Coherence Community projects provide example implementations for commonly used design patterns based on Oracle Coherence.

4

Integrating Coherence Applications with Coherence*Web

You can configure applications running under Coherence*Web so that they can share Coherence cache and session information.
If you are new to Coherence*Web, see Understanding Coherence*Web in *Administering HTTP Session Management with Oracle Coherence*Web*.

This chapter includes the following section:

Merging Coherence Cache and Session Information

In Coherence, the cache configuration deployment descriptor provides detailed information about the various caches that can be used by applications within a cluster. Coherence provides a sample cache configuration deployment descriptor, named `coherence-cache-config.xml`, in the root of the `coherence.jar` library. In Coherence*Web, the session cache configuration deployment descriptor provides detailed information about the caches, services, and attributes used by HTTP session management. Coherence*Web provides a sample session cache configuration deployment descriptor, named `default-session-cache-config.xml`, in the `coherence-web.jar` library. You can use this file as the basis for any custom session cache configuration file you may need to write.
At run time, Coherence uses the first `coherence-cache-config.xml` file that is found in the classpath, and it must precede the `coherence.jar` library; otherwise, the sample `coherence-cache-config.xml` file in the `coherence.jar` file is used.

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

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

1. Extract the session cache configuration file from the `coherence-web.jar` library.
2. Merge the cache information from the Coherence cache configuration file into the session cache configuration file.

   Note that in the cache scheme mappings in this file, you cannot use wildcards to specify cache names. You must provide, at least, a common prefix for application cache names.
3. Ensure that modified session cache configuration file is used by the Coherence members in the cluster.

   The cache and session configuration must be consistent across WebLogic Servers and Coherence cache servers.

5

Using Memcached Clients with Oracle Coherence

You can configure a memcached adapter to allow Coherence to be used as a distributed cache for memcached clients. A simple hello world client that is written using the spymemcached API is provided for demonstration purposes; howver any existing memcached client can be used to connect to Coherence.
This chapter includes the following sections:

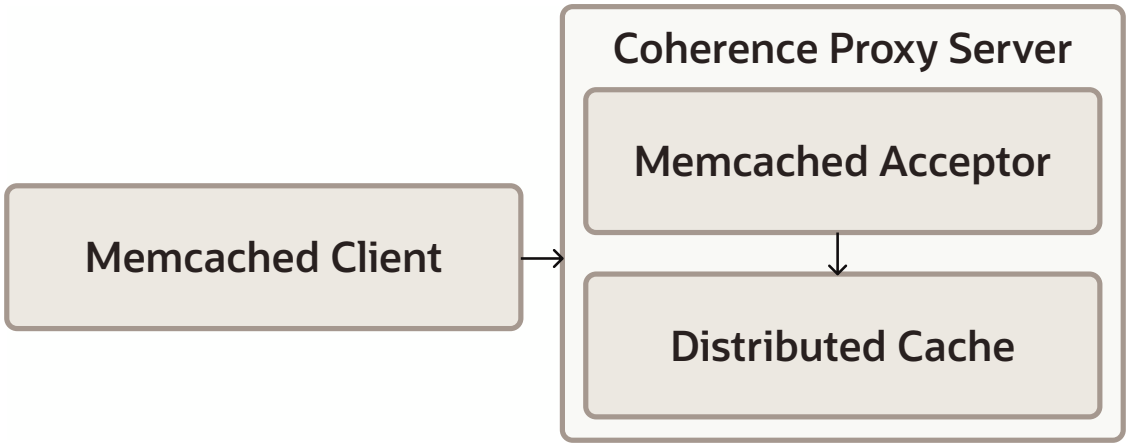
- Overview of the Oracle Coherence Memcached Adapter
- Setting Up the Memcached Adapter
  Memcached adapters are configured within a proxy service using a specific memcached acceptor. The acceptor configuration defines the socket address and the distributed cache for use by memcached clients.
- Connecting to the Memcached Adapter
- Securing Memcached Client Communication
- Sharing Data Between Memcached and Coherence Clients

Overview of the Oracle Coherence Memcached Adapter

The memcached adapter provides access to Coherence caches over the memcached binary protocol and allows Coherence to be used as a drop-in replacement for a memcached server. The adapter supports any memcached client API that supports the memcached binary protocol. This allows memcached clients that are written in many different programming languages to use Coherence.
The memcached adapter is located on a Coherence proxy server and is implemented as a Coherence*Extend-styled acceptor. Memcached clients connect to the acceptor, which manages the distributed cache operations on the cluster. The cache operations are performed as entry processor operations. The acceptor must first be enabled within a proxy service in order to interact with Coherence cached data. Additional features for securing memcached client communication and for sharing data with native Coherence clients are provided and can be configured as required.

Figure 5-1 shows a conceptual view of a memcached client connecting to the memcached acceptor located on a Coherence proxy server in order to use a distributed cache.

Figure 5-1 Conceptual View of a Memcached Client Connection



Setting Up the Memcached Adapter

Memcached adapters are configured within a proxy service using a specific memcached acceptor. The acceptor configuration defines the socket address and the distributed cache for use by memcached clients.

This section includes the following topics:

- Define the Memcached Adapter Socket Address
- Define Memcached Adapter Proxy Service

Define the Memcached Adapter Socket Address

The memcached adapter uses a socket address (IP, or DNS name, and port) for clients to connect to. The socket address is configured in an operational override configuration file using the `<address-provider>` element. The address is then referenced from a proxy service definition using the configured `id` attribute. See address-provider in *Developing Applications with Oracle Coherence*.

The following example configures a socket address and uses `198.168.1.5` for the IP address, `9099` for the port, and `memcached` for the ID.

```
...
<cluster-config>
   <address-providers>
      <address-provider id="memcached">
         <socket-address>
            <address>198.168.1.5</address>
            <port>9099</port>
         </socket-address>
      </address-provider>
   </address-providers>
</cluster-config>
...
```

Define Memcached Adapter Proxy Service

A proxy service allows remote clients to interact with the caching services of a Coherence cluster without becoming cluster members. A proxy service for the memcached adapter includes a specific memcached acceptor that accepts memcached client requests on a defined socket address and then delegates the requests to a distributed cache.

> **Note:**
>
> The memcached adapter can only use a distributed cache.

To create a proxy service for memcached clients, edit the cache configuration file and add a `<proxy-scheme>` element and include the `<memcached-acceptor>` element within the `<acceptor-config>` element. The `<memcached-acceptor>` element must include the name of the cache to use and a reference to an address provider definition that defines the socket address to listen to for memcached client communication. See memcached-acceptor in *Developing Applications with Oracle Coherence*.

The following example creates a proxy service and defines a memcached acceptor. The example references the address provider that was defined in Define the Memcached Adapter Socket Address.

```
...
<caching-schemes>
   <proxy-scheme>
      <service-name>MemcachedProxyService</service-name>
      <acceptor-config>
         <memcached-acceptor>
            <cache-name>hello-example</cache-name>
            <address-provider>memcached</address-provider>
         </memcached-acceptor>
      </acceptor-config>
      <autostart>true</autostart>
   </proxy-scheme>
</caching-schemes>
...
```

The cache name refers to the `hello-example` cache. The cache name must resolve to a distributed cache. The following example shows the definition of the `hello-example` cache and the distributed scheme to which it maps.

```
<?xml version="1.0"?>
<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
              xsi:schemaLocation=
                "http://xmlns.oracle.com/coherence/coherence-cache-config
                coherence-cache-config.xsd">

   <caching-scheme-mapping>
      <cache-mapping>
         <cache-name>hello-example</cache-name>
         <scheme-name>distributed</scheme-name>
      </cache-mapping>
   </caching-scheme-mapping>

   <caching-schemes>
```

```
      <distributed-scheme>
         <scheme-name>distributed</scheme-name>
         <service-name>MemcachedTest</service-name>
         <backing-map-scheme>
           <local-scheme/>
         </backing-map-scheme>
         <autostart>true</autostart>
      </distributed-scheme>

      <proxy-scheme>
         <service-name>MemcachedProxyService</service-name>
         <acceptor-config>
            <memcached-acceptor>
               <cache-name>hello-example</cache-name>
               <address-provider>memcached</address-provider>
            </memcached-acceptor>
         </acceptor-config>
         <autostart>true</autostart>
      </proxy-scheme>
   </caching-schemes>
</cache-config>
```

Connecting to the Memcached Adapter

Memcached clients must specify the address and port of a proxy service for the memcached adapter. The proxy service address is used in place of the memcached server address. Refer to your memcached client documentation for details on how to specify the address of a memcached server.

The following example shows a simple hello world client that uses the spymemcached client API to connect to the proxy service for the memcached adapter that was defined in Setting Up the Memcached Adapter.

```
import net.spy.memcached.AddrUtil;
import net.spy.memcached.BinaryConnectionFactory;
import net.spy.memcached.MemcachedClient;

public class MemcachedExample {
   public static void main(String[] args) throws Exception {
      String key = "k1";
      String value = "Hello World!";

      MemcachedClient c = new MemcachedClient(
         new BinaryConnectionFactory(),
         AddrUtil.getAddresses("198.168.1.5:9099"));

      c.add(key, 0, value);
      System.out.println((String)c.get(key));
      c.shutdown();
   }
}
```

Securing Memcached Client Communication

The memcached adapter can use both authentication and authorization to restrict access to cluster resources. Authentication support is provided for the SASL (Simple Authentication and Security Layer) plain authentication. Authorization is implemented using Oracle Coherence*Extend-styled authorization, which relies on interceptor classes that provide fine-grained access for cache service operations. The memcached adapter authentication and

authorization features reuses much of the existing security capabilities of Oracle Coherence: references are provided to existing content where applicable.
This section includes the following topics:

- Performing Memcached Client Authentication
- Performing Memcached Client Authorization

Performing Memcached Client Authentication

Memcached clients can use SASL plain authentication to provide a username and password when connecting to the memcached adapter. To use SASL plain authentication, you must create an `IdentityAsserter` implementation on the proxy. The memcached adapter calls the `IdentityAsserter` implementation and passes the `com.tangosol.net.security.UsernameAndPassword` object as a token. See Using Identity Tokens to Restrict Client Connections in *Securing Oracle Coherence*. Refer to your memcached client documentation for details on establishing a SASL plain connection.

In addition to an `IdentityAsserter` implementation, authentication must be enabled on a memcached adapter to use SASL plain authentication. To enable authentication, edit the proxy service definition in the cache configuration file and add a `<memcached-auth-method>` element, within the `<memcached-acceptor>` element, and set it to `plain`.

```
...
<caching-schemes>
   <proxy-scheme>
      <service-name>MemcachedProxyService</service-name>
      <acceptor-config>
         <memcached-acceptor>
            <cache-name>hello-example</cache-name>
            <memcached-auth-method>plain</memcached-auth-method>
            <address-provider>memcached</address-provider>
         </memcached-acceptor>
      </acceptor-config>
      <autostart>true</autostart>
   </proxy-scheme>
</caching-schemes>
...
```

Performing Memcached Client Authorization

The memcached adapter relies on the Oracle Coherence*Extend authorization framework to restrict which operations a memcached client performs on a cluster. See Implementing Extend Client Authorization in *Securing Oracle Coherence*.

Sharing Data Between Memcached and Coherence Clients

The memcached adapter stores entries in a cache using a binary format. If you intend to share the data with Coherence clients, then memcached clients must use a serialization format that Coherence clients also support. Coherence clients typically use Portable Object Format (POF), which is highlighted in this section. See Using Portable Object Format in *Developing Applications with Oracle Coherence*.
This section includes the following topics:

- Configuring POF for Memcached Clients
- Create a Memcached Client that Uses POF

Configuring POF for Memcached Clients

To configure POF for Memcached clients:

1. Edit the proxy service definition in the cache configuration file and add an `<interop-enabled>` element, within the `<memcached-acceptor>` element, and set it to `true`.

```
...
<proxy-scheme>
   <service-name>MemcachedProxyService</service-name>
   <acceptor-config>
      <memcached-acceptor>
         <cache-name>hello-example</cache-name>
         <interop-enabled>true</interop-enabled>
         <address-provider>memcached</address-provider>
      </memcached-acceptor>
   </acceptor-config>
   <autostart>true</autostart>
</proxy-scheme>
...
```

2. Enable POF on the distributed cache that is used by the memcached acceptor.

```
...
<distributed-scheme>
   <scheme-name>distributed</scheme-name>
   <service-name>MemcachedTest</service-name>
   <serializer>
      <instance>
         <class-name>com.tangosol.io.pof.ConfigurablePofContext</class-name>
         <init-params>
            <init-param>
               <param-type>String</param-type>
               <param-value>memcached-pof-config.xml</param-value>
            </init-param>
         </init-params>
      </instance>
   </serializer>
   <backing-map-scheme>
      <local-scheme/>
   </backing-map-scheme>
   <autostart>true</autostart>
</distributed-scheme>
```

3. Register POF types in the defined POF configuration file. For the above example, the POF configuration file is named `memcached-pof-config.xml`. The file must be found on the classpath before the `coherence.jar` file. The following example defines a POF user type for the `PofUser` object:

```
<?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
   coherence-pof-config.xsd">
   <user-type-list>
      <include>coherence-pof-config.xml</include>

      <!-- User types must be above 1000 -->
      <user-type>
         <type-id>1001</type-id>
```

```
            <class-name>memcached.PofUser</class-name>
          </user-type>

       </user-type-list>
    </pof-config>
```

Create a Memcached Client that Uses POF

Many memcached client libraries include the ability to plug in custom serializers. Refer to your memcached client documentation for details on how to plug in custom serializers. The following excerpt shows a spymemcached client that adds the `PofUser` object that was registered in step 3 and uses a spymemcached transcoder to plug in the POF serializer.

```
MemcachedClient client = m_client;
String key = "pofKey";
PofUser user = new PofUser("memcached", 1);
PofTranscoder<PofUser> tc = new PofTranscoder("memcached-pof-config.xml");

if (!client.set(key, 0, user, tc).get())
   {
      throw new Exception("failed to set value");
   }
```

The POF transcoder plug-in is defined as follows:

```
import com.tangosol.io.pof.ConfigurablePofContext;
import com.tangosol.util.Binary;
import com.tangosol.util.ExternalizableHelper;

import net.spy.memcached.CachedData;
import net.spy.memcached.compat.SpyObject;
import net.spy.memcached.transcoders.Transcoder;


public class PofTranscoder<T> extends SpyObject implements Transcoder<T>
    {

    public PofTranscoder(String sLocator)
        {
        m_ctx = new ConfigurablePofContext(sLocator);
        }

    @Override
    public boolean asyncDecode(CachedData arg0)
        {
        return Boolean.FALSE;
        }

    @Override
    public T decode(CachedData cachedData)
        {
        int nFlag = cachedData.getFlags();
        Binary bin = new Binary(cachedData.getData());
        return (T) ExternalizableHelper.fromBinary(bin, m_ctx);
        }

    @Override
    public CachedData encode(Object obj)
        {

        byte[] oValue = ExternalizableHelper.toByteArray(obj, m_ctx);
```

```
    return new CachedData(FLAG, oValue, CachedData.MAX_SIZE);
    }

@Override
public int getMaxSize()
    {
    return CachedData.MAX_SIZE;
    }

protected ConfigurablePofContext m_ctx;

protected static final int       FLAG = 4;
```

6

Integrating Spring with Coherence

Oracle Coherence can be integrated with Spring, which is a platform for building and running Java-based enterprise applications.

If you are interested in using Coherence with Spring, see the Coherence Spring Integration project that is part of the Coherence Community. Coherence Community projects provide example implementations for commonly used design patterns based on Oracle Coherence.

7

Using Coherence MicroProfile Configuration

Coherence MicroProfile (MP) Configuration provides support for Eclipse MicroProfile Configuration within Coherence cluster members. See Eclipse MicroProfile Configuration. Coherence MP Configuration enables you to configure various Coherence parameters from the values specified in any of the supported configuration sources, and to use Coherence cache as another, mutable configuration source.
This chapter includes the following topics:

- Enabling the Use of Coherence MicroProfile Configuration
  To use Coherence MP Configuration, you should first declare it as a dependency in the `pom.xml` file.
- Configuring Coherence Using MP Configuration
- Using Coherence Cache as a Configuration Source
  Coherence MP Configuration also provides an implementation of the Eclipse MP Configuration `ConfigSource` interface, which enables you to store configuration parameters in a Coherence cache.
- Examples Using Helidon MicroProfile with Coherence
  There are a number of open source example applications that demonstrate using Coherence MicroProfile integration with Helidon.

Enabling the Use of Coherence MicroProfile Configuration

To use Coherence MP Configuration, you should first declare it as a dependency in the `pom.xml` file.

You can declare Coherence MP Configuration as follows:

```
<dependency>
    <groupId>${coherence.groupId}</groupId>
    <artifactId>coherence-mp-config</artifactId>
    <version>${coherence.version}</version>
</dependency>
```

You will also need an implementation of the Eclipse MP Configuration specification as a dependency. For example, if you are using Helidon, add the following to the `pom.xml` file:

```
<dependency>
    <groupId>io.helidon.microprofile.config</groupId>
    <artifactId>helidon-microprofile-config</artifactId>
    <version>2.5.0</version>
</dependency>

<!-- optional: add it if you want YAML config file support -->
<dependency>
    <groupId>io.helidon.config</groupId>
    <artifactId>helidon-config-yaml</artifactId>
```

```
    <version>2.5.0</version>
</dependency>
```

Configuring Coherence Using MP Configuration

Coherence provides a number of configuration properties that you can use to define certain attributes or to customize cluster member behavior at runtime.

For example, you can define attributes such as cluster and role name, as well as define whether a cluster member should or should not store data, through the use of system properties:

```
-Dcoherence.cluster=MyCluster -Dcoherence.role=Proxy -
Dcoherence.distributed.localstorage=false
```

You can also define most of these attributes within the operational or cache configuration file. For example, you could define first two attributes, cluster name and role, within the operational configuration override file:

```
<cluster-config>
  <member-identity>
    <cluster-name>MyCluster</cluster-name>
    <role-name>Proxy</role-name>
  </member-identity>
</cluster-config>
```

While these two options are more than enough in most cases, there are some issues with them being the **only** way to configure Coherence:

- When you are using one of the Eclipse MicroProfile implementations, such as Helidon (see Helidon as the foundation of your application, Oracle recommends that you define some of Coherence configuration parameters along with the other configuration parameters, and not in a separate file or through system properties.
- In some environments, such as Kubernetes, Java system properties are cumbersome to use, and environment variables are a preferred way of passing configuration properties to containers.

Unfortunately, neither of the two use cases above is supported out-of-the-box. Coherence MP Configuration is designed to fill this gap.

As long as you have `coherence-mp-config` and an implementation of Eclipse MP Configuration specification to your class path, Coherence will use any of the standard or custom configuration sources to resolve various configuration options it understands.

Standard configuration sources in MP Configuration include the `META-INF/microprofile-config.properties` file, if present in the class path; environment variables; and system properties (in that order, with the properties in the latter overriding the ones from the former). These configuration sources directly address the second use case mentioned above, and allow you to specify Coherence configuration options through environment variables within the Kubernetes YAML files. For example:

```
containers:
    - name: my-app
      image: my-company/my-app:1.0.0
```

```
      env:
       - name: COHERENCE_CLUSTER
         value: "MyCluster"
       - name: COHERENCE_ROLE
         value: "Proxy"
       - name: COHERENCE_DISTRIBUTED_LOCALSTORAGE
         value: "false"
```

The above is just an example. If you are running the Coherence cluster in Kubernetes, you should really be using Coherence Operator instead, as it will make both the configuration and the operation of the Coherence cluster much easier.

You can also specify the Coherence configuration properties along with the other configuration properties of your application, which will enable you to keep everything in one place, and not scattered across many files. For example, if you are writing a Helidon application, you can simply add the `coherence` section to the `application.yaml` file:

```
coherence:
  cluster: MyCluster
  role: Proxy
  distributed:
    localstorage: false
```

Using Coherence Cache as a Configuration Source

Coherence MP Configuration also provides an implementation of the Eclipse MP Configuration `ConfigSource` interface, which enables you to store configuration parameters in a Coherence cache.

This feature has several benefits:

- Unlike pretty much all of the default configuration sources, which are static, configuration options stored in a Coherence cache can be modified without forcing you to rebuild your application JARs or Docker images.
- You can change the value in one place, and it will automatically be visible and up to date on all the members.

While the features above give you incredible amount of flexibility, it may not always be desirable. Therefore, this feature is disabled by default.

If you want to enable it, you should do so explicitly by registering `CoherenceConfigSource` as a global interceptor in the cache configuration file:

```
<cache-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
              xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-
cache-config coherence-cache-config.xsd">

  <interceptors>
    <interceptor>
      <instance>
        <class-name>com.oracle.coherence.mp.config.CoherenceConfigSource</
class-name>
      </instance>
    </interceptor>
```

```
  </interceptors>

  <!-- your cache mappings and schemes... -->

</cache-config>
```

After you enable the feature, `CoherenceConfigSource` is activated as soon as the cache factory is initialized, and injected into the list of available config sources for your application to use through the standard MP Configuration APIs.

By default, it will be configured with a priority (ordinal) of 500, making it of a higher priority than all the standard configuration sources, thus allowing you to override the values provided through configuration files, environment variables, and system properties. However, you have full control over that behavior and can specify a different ordinal through the `coherence.mp.config.source.ordinal` configuration property.

Examples Using Helidon MicroProfile with Coherence

There are a number of open source example applications that demonstrate using Coherence MicroProfile integration with Helidon.

For more information about example applications, see the following items:

- **Helidon Sock Shop**

  This project is an implementation of a stateful, microservices based application that uses Oracle Coherence Community Edition as a scalable embedded data store, and Helidon MP as an application framework

  If you are interested in using this, see the Coherence Helidon Sock Shop sample.

- **Todo List Example**

  This repository contains a set of simple task management examples written in various languages to showcase Coherence Community Edition.

  In particular, the Java directory showcases how to integrate Coherence with Helidon MicroProfile.

  If you are interested in using this, see the Todo List example.

8

Using Coherence MicroProfile Health

Coherence MicroProfile (MP) Health provides support for Eclipse MicroProfile Health within the Coherence cluster members.

For more information about MicroProfile Health, see the following documentation:

- Using the Health Check API in *Managing Oracle Coherence*
- MicroProfile Health

Coherence MP Health is a very simple module that enables you to publish Coherence health checks into the MicroProfile Health Check Registries available at runtime.

This chapter includes the following topic:

- Enabling the Use of Coherence MP Health
  To use Coherence MP Health, you should first declare it as a dependency in the project's `pom.xml` file.

Enabling the Use of Coherence MP Health

To use Coherence MP Health, you should first declare it as a dependency in the project's `pom.xml` file.

You can declare Coherence MP Health as follows:

```
<dependency>
    <groupId>${coherence.groupId}</groupId>
    <artifactId>coherence-mp-health</artifactId>
    <version>${coherence.version}</version>
</dependency>
```

Where:

- `${coherence.groupId}` is the Maven group ID for the Coherence edition being used: `com.oracle.coherence` for the commercial edition or `com.oracle.coherence.ce` for the community edition.
- `${coherence.version}` is the version of Coherence you are using.

After the module becomes available in the class path, the Coherence `HealthCheck` producer CDI bean is automatically discovered and registered as a MicroProfile health check provider. The Coherence health checks then become available through any health endpoints served by the application and is included in started, readiness, and liveness checks.

9

Using Coherence MicroProfile Metrics

Coherence MicroProfile (MP) Metrics provides support for Eclipse MicroProfile Metrics within the Coherence cluster members. See MicroProfile Metrics. Coherence MP Metrics is a very simple module that enables you to publish Coherence metrics into the MicroProfile Metric Registries available at runtime, and adds Coherence-specific tags to all the metrics published within the process, to distinguish them on the monitoring server, such as Prometheus.
This chapter includes the following topics:

- Enabling the Use of Coherence MP Metrics
  To use Coherence MP Metrics, you should first declare it as a dependency in the `pom.xml` file.
- Coherence Global Tags

Enabling the Use of Coherence MP Metrics

To use Coherence MP Metrics, you should first declare it as a dependency in the `pom.xml` file.

You can declare Coherence MP Metrics as follows:

```
<dependency>
    <groupId>${coherence.groupId}</groupId>
    <artifactId>coherence-mp-metrics</artifactId>
    <version>${coherence.version}</version>
</dependency>
```

After the module becomes available in the class path, Coherence will discover the `MpMetricRegistryAdapter` service it provides, and use it to publish all standard Coherence metrics to the vendor registry, and any user-defined application metrics to the application registry.

All the metrics will be published as gauges, because they represent point-in-time values of various MBean attributes.

Coherence Global Tags

There could be hundreds of members in a Coherence cluster, with each member publishing potentially the same set of metrics. There could also be many Coherence clusters in the environment, possibly publishing to the same monitoring server instance. To help distinguish metrics coming from different clusters, as well as from different members of the same cluster, Coherence MP Metrics automatically adds several tags to all the metrics published within the process.

Table 9-1 Tags Used by Coherence MP Metrics

Tag Name	Tag Value
`cluster`	The name of the cluster.

Table 9-1 (Cont.) Tags Used by Coherence MP Metrics

Tag Name	Tag Value
`site`	The site to which the member belongs (if set).
`machine`	The machine on which the member is present (if set).
`member`	The name of the member (if set).
`node_id`	The node ID of the member.
`role`	The member's role.

Tagging ensures that the metrics published by one member do not collide with and overwrite the metrics published by other members. Tagging also helps you query and aggregate metrics based on the values of the tags above, if required.

10

Enabling ECID in Coherence Logs

Oracle Coherence can use an Execution Context ID (ECID). This globally unique ID can be attached to requests between Oracle components. The ECID allows you to track log messages pertaining to the same request when multiple requests are processed in parallel.
Coherence logs will include ECID only if the client already has an activated ECID prior to calling Coherence operations. The ECID may be passed from another component or obtained in the client code. To activate the context, use the `get` and `activate` methods on the `oracle.dms.context.ExecutionContext` interface in the Coherence client code. The ECID will be attached to the executing thread. Use the `deactivate` method to release the context, for example:

Example 10-1 Using a DMS Context in Coherence Client Code

```
...
// Get the context associated with this thread
ExecutionContext ctx = ExecutionContext.get();
ctx.activate();
...
set additional execution context values (optional)
perform some cache operations
...
// Release the context
ctx.deactivate();
...
```

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

To include the ECID in a Coherence log message, see Changing the Log Message Format in *Developing Applications with Oracle Coherence*.