Oracle® Fusion Middleware

Solution Guide for Oracle TopLink

11*g* Release 1 (11.1.1)

E25034-02

March 2012

This document describes a number of scenarios, or use cases, that illustrate TopLink features and typical TopLink development processes.

ORACLE

Oracle Fusion Middleware Solution Guide for Oracle TopLink, 11*g* Release 1 (11.1.1)

E25034-02

Contents

5 Scaling TopLink Applications in Clusters

6 Providing Software as a Service

7 Mapping JPA to XML

8 Testing TopLink JPA Outside a Container

9 Enhancing TopLink Performance

10 Migrating From Hibernate to TopLink

A Installing Oracle TopLink

Preface

Oracle TopLink delivers a proven standards-based enterprise Java solution for all of your relational and XML persistence needs based on high performance and scalability, developer productivity, and flexibility in architecture and design.

Audience

A variety of engineers use Oracle TopLink. Users of Oracle TopLink are expected to be proficient in the use of technologies and services related to Oracle TopLink (for example, JPA). This document does not include details about related common tasks, but focuses on Oracle TopLink functionality.

Application Developers--Users who want to develop applications using any of the following technologies for persistence services:

- Java Persistence Architecture (JPA) 2.x plus TopLink JPA extensions
- Java Architecture for XML Binding 2.x (JAXB) plus TopLink Object-XML extensions
- TopLink Database Web Services (DBWS)

Developers should be familiar with the concepts and programming practices of Java SE and Java EE.

Developers using TopLink JPA should be familiar with the concepts and programming practices of JPA 2.0, as specified in the Java Persistence Architecture 2.0 specification at `http://jcp.org/en/jsr/detail?id=317`.

Developers using TopLink Object-XML should be familiar with the concepts and programming practices of JAXB 2.0, as specified in the Java Architecture for XML Binding 2.0 specification at `http://jcp.org/aboutJava/communityprocess/pfd/jsr222/index.html`.

Developers using TopLink DBWS should be familiar with the concepts and programming practices of JAX-WS 2.0, as specified in the Java API for XML-Based Web Services 2.0 specification at `http://jcp.org/aboutJava/communityprocess/pfd/jsr222/index.html`

Administrator/Deployer--Users who want to deploy and manage applications using TopLink persistence technologies. These users should be familiar with basic operations of the chosen application server.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the following documents:

- *Oracle Fusion Middleware Oracle TopLink Concepts*
- *Oracle Fusion Middleware Java API Reference for Oracle TopLink*
- EclipseLink Documentation Center at http://wiki.eclipse.org/EclipseLink/UserGuide
- "Oracle TopLink" in *Oracle Fusion Middleware Release Notes for Linux x86*

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 Introduction

Oracle TopLink is a powerful and flexible Java persistence framework for storing Java objects in a relational database or for converting Java objects to XML documents. TopLink provides APIs and a run-time environment for implementing the persistence layer of Java EE applications.

TopLink is based on (and includes) EclipseLink, the open source persistence framework from the Eclipse Foundation. For more information about the EclipseLink project, see "Eclipse Persistence Services Project (EclipseLink) wiki home" at http://wiki.eclipse.org/EclipseLink. For the EclipseLink Documentation, Center see http://wiki.eclipse.org/EclipseLink/Documentation_Center.

1.1 About This Book

This book, *Solutions Guide for Oracle TopLink*, documents a number of scenarios, or use cases, that illustrate TopLink features and typical TopLink development processes. These are not tutorials that lead you step-by-step through every task required to complete a project. Rather, they document general processes and key details for solving particular problems and then provide links to other documentation for more information.

1.2 What You Need to Know First

To make good use of this documentation, you should already be familiar with the following:

- The concepts and programming practices of Java SE and Java EE. In the current release, TopLink supports Java EE 6. However, the current release of WebLogic Server (documented in several use cases) only supports Java EE 5. For more information, see the following:

  Java

  - Java home page: http://www.oracle.com/us/technologies/java/index.html
  - Java EE 5 Tutorial: http://download.oracle.com/javaee/5/tutorial/doc/bnbpy.html
  - Java EE 6 Tutorial: http://download.oracle.com/javaee/6/tutorial/doc/bnbpy.html

  - Any of the thousands of books and web sites devoted to Java.

  Oracle Java EE Application Servers

  - Oracle WebLogic Server home page: http://www.oracle.com/technetwork/middleware/weblogic/overview/index.html
  - Oracle Glassfish Server home page: http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html

  Oracle Java EE Integrated Development Environments

  - Oracle JDeveloper: http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html
  - Oracle Enterprise Pack for Eclipse (OEPE): http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html

- TopLink is based on EclipseLink from the Eclipse Foundation. See the following:
  - EclipseLink project home: http://wiki.eclipse.org/EclipseLink
  - EclipseLink Documentation Center: http://wiki.eclipse.org/EclipseLink/Documentation_Center
- If you are working with TopLink Java Persistence Architecture, you should be familiar with the concepts and programming practices of JPA 2.0, as specified in the *Java Persistence API, Version 2.0* specification at http://jcp.org/en/jsr/detail?id=317.
- If you are working with TopLink MOXy, you should be familiar with the concepts and programming practices of JAXB 2.0, as specified in the *The Java Architecture for XML Binding (JAXB) 2.0* specification at http://jcp.org/en/jsr/detail?id=222.
- Developers using TopLink DBWS should be familiar with the concepts and programming practices of JAX-WS 2.0, as specified in the Java *API for XML-Based Web Services (JAX-WS) 2.0* specification at http://jcp.org/en/jsr/detail?id=224.
- Support for RESTful (Representational State Transfer) Web Services in the Java Platform at http://jcp.org/en/jsr/detail?id=311

1.3 The Use Cases

The use cases documented in this book are as follows:

- Chapter 2, "Using TopLink with WebLogic Server" - How to use TopLink as the persistence provider for TopLink Java Persistence Architecture (JPA) 2.0 applications deployed to WebLogic Server.
- Chapter 3, "Using TopLink with GlassFish Server" - How to use TopLink as the persistence provider for TopLink Java Persistence Architecture (JPA) 2.0 applications deployed to GlassFish Server.
- Chapter 4, "Using Multiple Databases with a Composite Persistence Unit" - How to expose multiple persistence units (each with unique sets of entity types) as a single persistence context.

- Chapter 5, "Scaling TopLink Applications in Clusters" - How to configure TopLink applications to ensure scalability in clustered application server environments.
- Chapter 6, "Providing Software as a Service" - How to make JPA entities or JAXB beans extensible; how to use multi-tenancy; and how to use an external metadata source.
- Chapter 7, "Mapping JPA to XML" - How to map JPA entities to XML using TopLink MOXy.
- Chapter 8, "Testing TopLink JPA Outside a Container" - How to test your TopLink application outside the container.
- Chapter 9, "Enhancing TopLink Performance" - Getting the best performance out of Oracle TopLink.
- Section 10, "Migrating From Hibernate to TopLink" - How to migrate applications from using Hibernate JPA to using TopLink JPA.

2

Using TopLink with WebLogic Server

This chapter introduces how TopLink can be used as the persistence provider for TopLink Java Persistence API (JPA) 2.0 applications deployed to WebLogic Server.

This chapter contains the following sections:

2.1 Understanding TopLink and WebLogic Server

WebLogic Server is a scalable, enterprise-ready Java Platform, Enterprise Edition (Java EE) application server. WebLogic Server's complete implementation of the Java EE 5.0 specification provides a standard set of APIs for creating distributed Java applications that can access a wide variety of services, such as databases, messaging services, and connections to external enterprise systems. In addition to the Java EE implementation, WebLogic Server enables enterprises to deploy mission-critical applications in a robust, secure, highly available, and scalable environment. These features allow enterprises to configure clusters of WebLogic Server instances to distribute load, and provide extra capacity in case of hardware or other failures. For more details about these and other WebLogic Server features, see *Introduction to WebLogic Server*.

2.1.1 Advantages to Using TopLink with WebLogic Server

While WebLogic Server can use other persistence providers and TopLink can be used with other application servers, using WebLogic Server with TopLink provides a number of advantages:

- TopLink is included in all WebLogic Server distributions, and WebLogic Server can be configured so that TopLink is the default persistence provider for WebLogic Server domains, with support for JPA 2.0. See Section 2.3.1, "Task 1: Set TopLink as the Default JPA Provider (WebLogic Server 11g),"and Section 2.3.2, "Task 2: Apply the Patch to Support JPA 2.0 in WebLogic Server 11g."
- Oracle WebLogic Suite includes Oracle Coherence, which is a Java-based in-memory data-grid product that provides data caching, data replication, and distributed computing services. WebLogic Server and Coherence are tightly integrated and allow applications to use Coherence data caches. WebLogic Server applications can use TopLink Grid, which is an integration between TopLink and Coherence that allows TopLink to use Coherence as a level 2 (L2) cache and persistence layer for entities. How to use these integrations is beyond the scope of this documentation. See *Oracle Coherence Developer's Guide* and *Oracle Fusion*

Middleware Integration Guide for Oracle TopLink with Coherence Grid for more information.

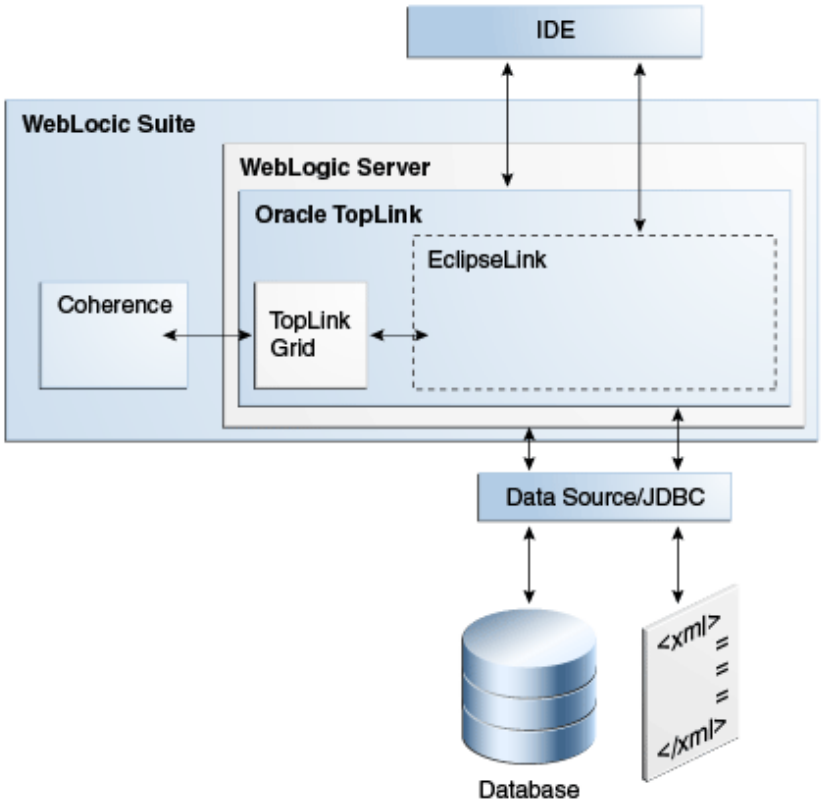
> **Note:** You can also obtain Coherence as a separately-licensed product to use WebLogic Server Standard Edition and WebLogic Server Enterprise Edition. See the links above for more information.

- TopLink logging integration in WebLogic Server provides a comprehensive, integrated logging infrastructure. See Section 2.3.5, "Task 5: Use or Reconfigure the Logging Integration."
- WebLogic Server supports the Oracle Application Framework (ADF), an end-to-end Java EE framework, based on Struts and JavaServer Faces (JSF). ADF simplifies application development by providing infrastructure services and a visual and declarative development experience. TopLink and ADF together provide a complete Java EE application infrastructure. How to use ADF is beyond the scope of this documentation. See *Developing Fusion Web Applications with Oracle Application Development Framework*.
- WebLogic Server, TopLink, and ADF are all integrated with JDeveloper, Oracle's integrated development environment (IDE) that provides end-to-end support for modeling, developing, debugging, optimizing, and deploying Java EE applications, including applications that use TopLink as the persistence provider and that are deployed to WebLogic Server. How to use JDeveloper is beyond the scope of this documentation. See `http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html` for general information about JDeveloper. For information about JDeveloper tasks, see the JDeveloper online help in the JDeveloper IDE.

2.1.2 The Relationship of TopLink to Other Fusion Middleware Products

Figure 2–1 shows how WebLogic Server and TopLink are related to and used with other Oracle products. You can: use JDeveloper (or Oracle Enterprise Pack for Eclipse or NetBeans) to develop Java EE applications; use TopLink as the persistence provider; use Oracle Coherence (via TopLink Grid integration) for data caching, data replication and distributed computing services; use WebLogic as the application server; and use the Oracle database for persisting data from TopLink JPA applications or XML for persisting data from TopLink MOXy applications.

Figure 2–1 Relationship of WebLogic Server, TopLink, and Related Products



2.2 What You Need to Start

To develop and deploy TopLink applications to Oracle WebLogic Server, you need:

- WebLogic Server. This documentation is based on Oracle WebLogic Server release 11gR1 (10.3.6).

  For more information and downloads, see http://www.oracle.com/technetwork/middleware/weblogic/overview/index.html on the Oracle Technology Network.

- Any compliant JDBC database including Oracle, Oracle Express, MySQL, etc.

  For the Oracle database, see http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html. For Oracle Express Edition, see http://www.oracle.com/technetwork/database/express-edition/overview/index.html. For MySQL, see http://www.oracle.com/us/products/mysql/index.html.

- While it is not required, you may want to use a Java development environment (IDE) for convenience during development. For example JDeveloper, Oracle Enterprise Pack for Eclipse (OEPE), and Oracle NetBeans all provide sophisticated Java EE development tools. Both JDeveloper and OEPE include embedded versions of WebLogic Server, although this documentation describes a standalone instance of WebLogic Server.

  For JDeveloper, see http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html. For OEPE, see

http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html. For NetBeans, see http://www.oracle.com/us/products/tools/050845.html.

2.3 Main Tasks

To run TopLink JPA applications in WebLogic server, you must configure WebLogic Server and coordinate certain settings in the server and your application, as described in the following steps.

- Task 1: Set TopLink as the Default JPA Provider (WebLogic Server 11g)
- Task 2: Apply the Patch to Support JPA 2.0 in WebLogic Server 11g
- Task 3: Update the Version of EclipseLink in WebLogic Server
- Task 4: Configure JMX MBean Extensions in WebLogic Server
- Task 5: Use or Reconfigure the Logging Integration
- Task 6: Add Persistence to Your Java Application Using TopLink
- Task 7: Configure a Data Source
- Task 8: Extend the Domain to Use Advanced Oracle Database Features
- Task 10: Start WebLogic Server and Deploy the Application
- Task 11: Run the Application
- Task 12: Configure and Monitor Persistence Settings in WebLogic Server

2.3.1 Task 1: Set TopLink as the Default JPA Provider (WebLogic Server 11*g*)

You can specify in a JPA application's `persistence.xml` file which JPA persistence provider to use for each entity. However, if no persistence provider is specified, the domain-wide default provider specified in WebLogic Server is used.

> **Note:** Oracle TopLink is the default JPA persistence provider in WebLogic Server 12*c*, so you do not have to do anything further to use it as the default provider in that release and later.

Oracle TopLink is not set as the default JPA persistence provider in WebLogic Server 11*g* (10.3.*n*), so you must set it to be the default provider in those releases.

> **Note:** The reason that TopLink is not the default persistence provider is to maintain backwards-compatibility with previous versions of WebLogic Server 10.3. This allows you to upgrade your version of WebLogic Server to future 10.3 patch sets, without changing the persistence provider.

Changing the default provider does not affect applications that are already deployed. The setting takes effect when the server is restarted or the application is manually redeployed.

To specify TopLink as the default JPA provider in the WebLogic Server Administration Console:

1. Start WebLogic Server and start the Administration Console.

2. If you have not already done so, in the Change Center of the Administration Console, click **Lock & Edit**.
3. In the left pane of the Console, under Domain Structure, select the domain name.
4. Select **Configuration > General > JPA**.
5. From the **Default JPA Provider** list, select **TopLink**.
6. Click **Save**.
7. To activate these changes, in the Change Center of the Administration Console, click **Activate Changes**.

2.3.2 Task 2: Apply the Patch to Support JPA 2.0 in WebLogic Server 11*g*

WebLogic Server 11gR1 (10.3.6) supports JPA 1.0 by default. However, you can apply a patch to provide support for JPA 2.0. For complete instructions, see "Using JPA 2.0 with TopLink in WebLogic Server" in *Programming Enterprise JavaBeans for Oracle WebLogic Server*.

Note: JPA 2.0 is backwards compatible so it can support applications that use the JPA 1.0 API.

Note: WebLogic Server 12*c* supports JPA 2.0 by default.

2.3.3 Task 3: Update the Version of EclipseLink in WebLogic Server

TopLink includes all the EclipseLink libraries, which provide the support for JPA, MOXy, DBWS, and other persistence and transformation services. The version of EclipseLink in TopLink may not be the most current available from the Eclipse Foundation. However, you can upgrade the EclipseLink version by using the WebLogic Server `FilteringClassLoader` and the shared library feature.

For what is supported in various releases, see the following:

- "Oracle TopLink: JPA Certification" at `http://www.oracle.com/technetwork/middleware/ias/jpa-082702.html#eclipselink`
- "Oracle TopLink and WebLogic Support" at `http://www.oracle.com/technetwork/middleware/ias/weblogic-086699.html`

The `FilteringClassLoader` provides a mechanism for configuring deployment descriptors to specify that certain packages are always loaded from the application, rather than being loaded by the system classloader. You can use this mechanism to specify that a newer version of EclipseLink be used by an application. For more information about filtering classloader, see "Using a Filtering Classloader" in *Developing Applications for Oracle WebLogic Server*.

A shared library is a Java EE module that can be shared by multiple enterprise applications. A shared library is deployed to a WebLogic Server target, and it can then be referenced by applications. Upon deployment, WebLogic Server merges the contents of the shared library with the application. In addition, because shared libraries can be packaged as standard Java EE archives, any descriptors are also merged with the application at deployment. For more information about WebLogic

Server shared libraries, see "Creating Shared Java EE Libraries and Optional Packages" in *Developing Applications for Oracle WebLogic Server*.

To update the version of EclipseLink used by applications deployed to WebLogic Server, do the following:

1. Download the `eclipselink-version_no.zip` for the EclipseLink version you want from the EclipseLink web site at `http://www.eclipse.org/eclipselink/downloads/index.php`.
2. Prepare the shared library as a standard Java EE Enterprise Archive (EAR), named, for example, `eclipselink-shared-lib.ear`, containing the following items:

   ```
   META-INF/weblogic-application.xml
   META-INF/application.xml
   lib/eclipselink.jar
   ```

   For more information about creating EARs, see, for example, "Creating and Configuring Web Applications" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.
3. In the application's `weblogic-application.xml` descriptor file, add a `prefer-application-packages` element, with the subelement `<package-name>org.eclipse.persistence.*</package-name>`, as shown below:

   ```
   <weblogic-application>
     <prefer-application-packages>
       <package-name>org.eclipse.persistence.*</package-name>
     </prefer-application-packages>
   </weblogic-application>
   ```

4. Create an `application.xml` file for the application. This file is necessary to support the runtime library merging. The minimum configuration is as follows:

   ```
   <application>
     <display-name>eclipselink-shared-lib</display-name>
     <module>
           <java></java>
     </module>
   </application>
   ```

5. Add extension name, specification version, and implementation version to the EAR's `META-INF/MANIFEST.MF` file. For example, if you are using ant, you can do the following:

   ```
   <target name="package" depends="prepare">
       <jar destfile="dist/${ant.project.name}.ear">
           <metainf dir="etc" includes="*.xml"/>
           <manifest>
               <attribute name="Extension-Name" value="eclipselink"/>
               <attribute name="Specification-Version" value="2.0"/>
               <attribute name="Implementation-Version" value="2.2.0"/>
           </manifest>
           <fileset dir="build" includes="**/*"/>
       </jar>
   </target>
   ```

At deployment time, WebLogic Server uses the attributes as metadata for the deployed shared-library.

The final EAR file should look like this:

```
META-INF/
META-INF/MANIFEST.MF
META-INF/application.xml
META-INF/weblogic-application.xml
lib/
lib/eclipselink.jar
```

6. Deploy `eclipselink-shared-lib.ear` to WebLogic Server. This results in a new library being available on the server, `eclipselink#2.0@2.2.0`.
7. In the `weblogic-application.xml` file of any applications that will use the updated version of EclipseLink, add a reference to the shared library, as shown below:

```
<weblogic-application>
    <library-ref>
        <library-name>eclipselink</library-name>
        <specification-version>2.0</specification-version>
        <implementation-version>2.2.0</implementation-version>
    </library-ref>
</weblogic-application>
```

2.3.4 Task 4: Configure JMX MBean Extensions in WebLogic Server

WebLogic Server uses Java Management Extensions (JMX) MBeans to configure, monitor, and manage WebLogic Server resources. For TopLink applications, MBeans are used to monitor and configure aspects of persistence units and are also used for logging.

> **Note:** When deployed to WebLogic Server, TopLink applications deploy MBeans when they connect to the database, not at deployment time.

For information about how MBeans are used in WebLogic Server, see *Developing Custom Management Utilities With JMX for Oracle WebLogic Server* and *Developing Manageable Applications With JMX for Oracle WebLogic Server*.

For information about TopLink logging in WebLogic Server, see Section 2.3.5, "Task 5: Use or Reconfigure the Logging Integration."

By default, when you deploy an EclipseLink application to Oracle WebLogic Server, the EclipseLink runtime deploys the following Java Management Extensions (JMX) MBeans to the Oracle WebLogic Server JMX service for each EclipseLink session:

- `org.eclipse.persistence.services.DevelopmentServices` - This class is meant to provide facilities for managing an EclipseLink session internal to EclipseLink over JMX.
- `org.eclipse.persistence.services.RuntimeServices` - This class is meant to provide facilities for managing an EclipseLink session external to EclipseLink over JMX.

Use the API that this JMX MBean exposes to access and configure your TopLink sessions at run time, using JMX code that you write, or to integrate your TopLink application with a third-party JMX management application, such as JConsole.

To obtain information about accessing information about custom MBeans, you must first enable anonymous lookup and then use a separate tool to access the MBean information.

To enable honeymooner lookup in the Administration Console, do the following:

1. If you have not already done so, in the Change Center of the Administration Console, click **Lock & Edit**.
2. In the left pane, select your domain to open the Settings page for your domain.
3. Expand **Security > General**.
4. Select **Anonymous Admin Lookup Enabled**.
5. To activate these changes, in the Change Center of the Administration Console, click **Activate Changes**.

For the instructions on how to access the MBean information using various tools, see "Accessing Custom MBeans," in *Developing Manageable Applications With JMX for Oracle WebLogic Server*.

For information about monitoring custom MBeans in the WebLogic Server Administration Console, see "Monitor Custom MBeans" in *Oracle WebLogic Server Administration Console Online Help*.

2.3.5 Task 5: Use or Reconfigure the Logging Integration

By default, TopLink logging is integrated into the WebLogic Server logging infrastructure. Details about the integration works and how to override it are described in the following sections. For detailed information about WebLogic Server logging, see the following:

- *Using Logging Services for Application Logging for Oracle WebLogic Server*
- *Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*
- The logging topics in *Oracle WebLogic Server Administration Console Online Help*

For information about configuring logging for JPA persistence units, see "How to Configure Logging" in the EclipseLink documentation at `http://wiki.eclipse.org/EclipseLink/Examples/JPA/Logging`.

2.3.5.1 How the Logging Integration Works

By default, the WebLogic Server logging implementation is injected into the persistence context, which results in all TopLink logging messages being output according to the WebLogic Server logging configuration.

As a result of this integration, TopLink logging levels are converted to WebLogic Server logging levels as shown in Table 2–1.

Table 2–1 Mapping of TopLink Logging Levels to WebLogic Server Logging Levels

TopLink Logging Levels	WebLogic Server Logging Levels
ALL, FINEST, FINER, FINE	DEBUG
CONFIG	INFO
INFO	NOTICE
WARNING	WARNING
SEVERE	ALERT

Table 2–1 (Cont.) Mapping of TopLink Logging Levels to WebLogic Server Logging

TopLink Logging Levels	WebLogic Server Logging Levels
OFF	OFF

WebLogic Server logging levels are mapped to TopLink levels as shown in Table 2–2.

Table 2–2 Mapping of WebLogic Server Logging Levels to TopLink Logging Levels

WebLogic Server Logging Levels	TopLink Logging Levels
TRACE, DEBUG	FINEST
INFO	CONFIG
NOTICE	INFO
WARNING	WARNING
ERROR, CRITICAL, ALERT	SEVERE
EMERGENCY, OFF	OFF

2.3.5.2 Viewing Persistence Unit Logging Levels in the Administration Console

You can see the TopLink logging level defined for the persistence unit in the Administration Console, as described in Section 2.3.11, "Task 12: Configure and Monitor Persistence Settings in WebLogic Server." However, be aware that this logging level, set in persistence.xml is overridden when the default WebLogic Server/ TopLink logging integration is used. For instructions on overriding the integration, see the next section, Section 2.3.5.3, "Overriding the Default Logging Integration."

When the default integration is used, the EJB logging options for persistence are mapped through and control TopLink's logging output in the Administration Console.

2.3.5.3 Overriding the Default Logging Integration

You set TopLink logging levels in the `persistence.xml` file. However, when you accept the default logging integration with WebLogic Sever, those settings are ignored, and the logging configuration set in WebLogic Server is used. The TopLink logging levels are used only when you use the native TopLink logging implementation.

You can override the default logging integration by setting `eclipselink.logging.logger` property name to a different setting, for example:

To enable the default TopLink logging set the `eclipselink.logging.logger` property as follows:

```
<property name="eclipselink.logging.logger" value="DefaultLogger"/>
```

You can also use `java.util.logging` to use a different logging implementation for TopLink message, for example `java.util.logging`:

```
<property name="eclipselink.logging.logger" value="JavaLogger"/>
```

2.3.5.4 Configuring WebLogic Server to Expose TopLink Logging

If you use the native TopLink logging implementation, you can still display TopLink logging messages in the WebLogic Server domain's log files by configuring WebLogic Server to redirect Java Virtual Machine (JVM) output to the registered log destinations.

For more information and instructions for configuring the redirect, see "Redirecting JVM Output" in *Configuring Log Files and Filtering Log Messages for Oracle WebLogic*

Server. To set this option in the Administration Console, see "Redirect JVM output" in *Oracle WebLogic Server Administration Console Online Help*.

2.3.5.5 Other Considerations

You should be aware of these other considerations:

- The message ID `2005000` is used for all TopLink log messages.
- Some logging messages handled by WebLogic Server's integrated logger may show up in the WebLogic Server console or the server log (depending on the settings of logging levels) during deployment, even though at runtime the application's entity manager factory will use only the TopLink logging infrastructure and only the TopLink logging settings.
- If you use a different version of EclipseLink than the version bundled in your WebLogic Server installation (by using a filtering classloader), trying to using the default integrated logging can lead to errors, due to classloading conflicts. To work around this issue, explicitly set the `eclipselink.logging.logger` property to something other than the integrated WebLogic Server logger.

2.3.6 Task 6: Add Persistence to Your Java Application Using TopLink

Using TopLink JPA to provide persistence for an application is the fundamental task presumed by all the other tasks described in this chapter; yet the actual JPA programming practice is mostly outside the scope of this documentation. WebLogic Server imposes no special requirements on your TopLink application, other than the details described in this chapter.

That is because TopLink is based on EclipseLink, the open source persistence project from the Eclipse Foundation. EclipseLink 2.n is the reference implementation of the *Java Persistence API, Version 2.0* specification. TopLink includes all of the EclipseLink jars, plus additional Oracle tools and features. Therefore, you can take advantage of the full range of features and functionality provided by JPA and EclipseLink when using TopLink to add the persistence layer to your Java applications.

This chapter describes features, settings, and tasks that are specific to using TopLink (runtime and API) with WebLogic Server. For information about developing, packaging, and deploying a Java application using JPA, see the following:

- The EclipseLink project wiki at `http://wiki.eclipse.org/EclipseLink`
- The EclipseLink Documentation Center at `http://wiki.eclipse.org/EclipseLink/Documentation_Center`
- The *Java Persistence API, Version 2.0* specification at `http://jcp.org/en/jsr/detail?id=317`
- "Part V, Persistence" in "The Java EE 5 Tutorial" at `http://download.oracle.com/javaee/5/tutorial/doc/bnbpy.html`
- Any third-party book that describes programming Java applications using JPA

For more information about TopLink features and concepts, see Chapter 1, "Introduction" and *Oracle Fusion Middleware Oracle TopLink Concepts*.

For related WebLogic Server programming topics, see any book in the WebLogic Server documentation set (see *Oracle Fusion Middleware Information Roadmap for Oracle WebLogic Server*), in particular the following:

- *Programming Enterprise JavaBeans for Oracle WebLogic Server*
- *Developing Applications for Oracle WebLogic Server*

- *Deploying Applications to Oracle WebLogic Server*
- *Programming JDBC for Oracle WebLogic Server*

2.3.7 Task 7: Configure a Data Source

In WebLogic Server, you configure database connectivity by adding JDBC data sources to WebLogic Server domains. Each WebLogic data source contains a pool of database connections. Applications look up the data source on the JNDI tree or in the local application context and then reserve a database connection with the `getConnection()` method. Data sources and their connection pools provide connection management processes to keep the system running efficiently.

For information on using JDBC with WebLogic Server, see the following:

- For complete documentation about working with JDBC in WebLogic Server, see *Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*, in particular:
  - "Configuring WebLogic JDBC Resources"
  - "Configuring JDBC Data Sources"
- For information about working with JDBC data sources in the WebLogic Administration Console, see the topics under "Configure JDBC" in the *Oracle WebLogic Server Administration Console Online Help*.

2.3.7.1 Ways to Configure Data Sources for JPA Applications

You can configure data sources for JPA applications deployed to WebLogic Server in a variety of ways, including the following:

- Configure a Globally-Scoped JTA Data Source
- Configure an Application-Scoped JTA Data Source
- Configure a non-JTA Data Source and Manage Transactions in the Application

2.3.7.2 Configure a Globally-Scoped JTA Data Source

The most common data source configuration is a globally-scoped JNDI data source, using JTA for transaction management, specified in the `persistence.xml` file. Configuration is straightforward, and multiple applications can access the data source.

Do the following:

- Create the Data Source in WebLogic Server
- Configure persisence.xml

2.3.7.2.1 Create the Data Source in WebLogic Server To set up a globally-scoped JNDI data source in the WebLogic Server Administration Console, do the following:

1. Create a new data source, as described in "Configure JDBC generic data sources" in the *Oracle WebLogic Server Administration Console Online Help*.

> **Note:** TopLink is compatible with any WebLogic Server data source that can be accessed using standard JNDI data source lookup by name. These instructions describe the wizard for a generic data source.

2. Enter values in the Create a New JDBC data source wizard, according to your needs. See "Create a JDBC Data Source" in *Oracle WebLogic Server Administration Console Online Help*. for more information.

> **Important:** The value used for **JNDI Name** (on the **JDBC Datasource Properties** page must be the same as the value used for the `<jta-data-source>` element in `persistence.xml`.

3. Configure connection pools, as described in "Configuring Connection Pool Features" in *Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*. The connection pool configuration can affect TopLink's ability to handle concurrent requests from the application. Properties should be tuned in the same way any connection pool would be tuned to optimize resources and application responsiveness.

2.3.7.2.2 Configure persisence.xml In the `persistence.xml` file, specify that the `transaction-type` is `JTA`, and provide the name of the data source in the `jta-data-source` element (prefaced by `jdbc/` or not), as shown in Example 2–1, below:

Example 2–1 persistence.xml File With JNDI Data Source Using JTA

```
...
  <persistence-unit name="example" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>JDBC Data Source-1</jta-data-source>
    <class>org.eclipse.persistence.example.jpa.server.business.Cell</class>
    <class>org.eclipse.persistence.example.jpa.server.business.CellAttribute</class>
  </persistence-unit>
```

2.3.7.3 Configure an Application-Scoped JTA Data Source

To configure an application-scoped data source that use JTA for transaction management, perform the following tasks:

1. "Specify That the Data Source Is Application-Scoped"
2. "Add the JDBC Module to the WebLogic Application Configuration"
3. "Configure the JPA Persistence Unit to Use the JTA Data Source"

2.3.7.3.1 Specify That the Data Source Is Application-Scoped To define an application-scoped data source, create a *name*`-jdbc.xml` JDBC module file and place it in the `META-INF` folder of the application's EAR archive. In that file, add `<scope>Application</scope>` to the `jdbc-data-source-params` section, as shown in Example 2–2.

Example 2–2 JDBC Data Source Defined in the name-jdbc.xml File

```
<jdbc-data-source ...>
...
  <jdbc-data-source-params>
    <jndi-name>SimpleAppScopedDS</jndi-name>
    <scope>Application</scope>
  </jdbc-data-source-params>
```

```
</jdbc-data-source>
```

> **Hint:** You can create the framework for the a *name*-jdbc.xml file by creating a globally scoped data source from the WebLogic Server Administration Console, as described in Section 2.3.7.2, "Configure a Globally-Scoped JTA Data Source," with these differences:
>
> - Do not associate the data source with a server
> - Add the <scope> element manually

For more information about JDBC module configuration files and jdbc-data-source (including <jdbc-driver-params> and <jdbc-connection-pool-params>), see "Configuring WebLogic JDBC Resources" in *Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*.

2.3.7.3.2 Add the JDBC Module to the WebLogic Application Configuration Add a reference to the JDBC module in the /META-INF/weblogic-application.xml application deployment descriptor in the EAR archive, as shown in Example 2–3. This registers the data source for use in the application.

Example 2–3 JDBC Module Defined in weblogic-application.xml

```
<wls:module>
  <wls:name>SimpleAppScopedDS</wls:name>
  <wls:type>JDBC</wls:type>
  <wls:path>META-INF/simple-jdbc.xml</wls:path>
</wls:module>
```

For more information about weblogic-application.xml application deployment descriptors, see "Understanding Application Deployment Descriptors" in *Deploying Applications to Oracle WebLogic Server* and "Enterprise Application Deployment Descriptor Elements" in *Developing Applications for Oracle WebLogic Server*.

2.3.7.3.3 Configure the JPA Persistence Unit to Use the JTA Data Source To make it possible for TopLink runtime to lazily look up an application-scoped data source, you must specify an additional data source property in the definition of the persistence unit in persistence.xml. For a JTA data source, add a fully-qualified javax.persistence.jtaDataSource property, with the value java:/app/jdbc/*data_source_name*, as shown in Example 2–4.

The values of the <jta-data-source> and <javax.persistence.jtaDataSource> properties must match.

Example 2–4 JTA Data Source Definition in persistence.xml

```
<?xml version="1.0" encoding="windows-1252" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="employee" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:/app/jdbc/SimpleAppScopedDS</jta-data-source>
    <properties>
      <property name="javax.persistence.jtaDataSource"
```

```
          value="java:/app/jdbc/SimpleAppScopedDS" />
    </properties>
  </persistence-unit>
</persistence>
```

2.3.7.4 Configure a non-JTA Data Source and Manage Transactions in the Application

To configure a non-JTA data source managed by the application, do the same as described in Section 2.3.7.3, "Configure an Application-Scoped JTA Data Source," but configure the JPA persistence unit to use a non-JTA data source by specifying a not-JTA data source, as shown in Example 2–5:

Example 2–5 non-JTA Data Source Definition in persistence.xml

```
<?xml version="1.0" encoding="windows-1252" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="employee" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <non-jta-data-source>OracleDS</non-jta-data-source>
    <properties>
      <property name="javax.persistence.nonJtaDataSource"
          value="OracleDS" />
    </properties>
  </persistence-unit>
</persistence>
```

Write the code in your application to handle the transactions, as described, for example, in "Transactions in EJB Applications" in *Programming JTA for Oracle WebLogic Server*.

2.3.7.5 Make Sure the Settings Match

Certain settings in the data source configuration must match certain settings in the application's `ejbModule/META-INF/persistence.xml` file. For the data source configuration in WebLogic Server, you can check the settings in the configuration files or in the Administration Console.

In the Administration Console, review settings as follows:

1. In the **Domain Structure** tree, expand **Services**, then select **Data Sources**.
2. On the Summary of JDBC Data Sources page, click the name of the data source.
3. On the **Settings for *data_source_name* > Configuration > General** page, find the value for **JNDI Name**, for example **localDS**. If using JTA, the name must match the `<jta-data-source>` in `persistence.xml`.
4. On the **Settings for *data_source_name* > Configuration > Connection Pool** page, review these settings:
   - The value for **URL** must match the `javax.persistence.jdbc.url` value in `persistence.xml`, for example, `jdbc:oracle:thin:@127.0.0.1:1521:XE`
   - The value for **Driver Class Name** must match the `javax.persistence.jdbc.driver` value in `persistence.xml`, for

example (for a JTA data source), `oracle.jdbc.xa.client.OracleXADataSource`.

The following examples show the values that must be shared in the domain's `config.xml` file and the application's `persistence.xml` file.

Example 2–6 Server Domain config.xml File

```
...
<domain...>
  <jdbc-system-resource>
    <name>localJTA</name>
    <target>AdminServer,ManagedServer_1,ManagedServer_2</target>
    <descriptor-file-name>jdbc/localJTA-4636-jdbc.xml</descriptor-file-name>
  </jdbc-system-resource>
</domain>
```

2.3.8 Task 8: Extend the Domain to Use Advanced Oracle Database Features

To fully support Oracle Spatial and Oracle XDB mapping capabilities (in both standalone Oracle WebLogic Server and the Oracle JDeveloper integrated WebLogic Server), you must use the `toplink-spatial-template.jar` and `toplink-xdb-template.jar` to extend the WebLogic Server domain to support Oracle Spatial and XDB, respectively.

To extend your WebLogic Server domain:

1. Download the `toplink-spatial-template.jar` (to support Oracle Spatial) and `toplink-xdb-template.jar` (to support Oracle XDB) files from:
   - http://download.oracle.com/otn/java/toplink/111110/toplink-spatial-template.jar
   - http://download.oracle.com/otn/java/toplink/111110/toplink-xdb-template.jar
2. Copy the files, as shown in Table 2–3 and Table 2–4:

Table 2–3 File to support Oracle Spatial

File	From...	To...
`sdoapi.jar`	*ORACLE_DATABASE_HOME*`/md/jlib`	*WL_HOME*`/server/lib`

Table 2–4 Files to support Oracle XDB

File	From...	To...
`xdb.jar`	*ORACLE_DATABASE_HOME*`/rdbms/jlib`	*WL_HOME*`/server/lib`
`xml.jar`	*ORACLE_DATABASE_HOME*`/lib`	*WL_HOME*`/server/lib`
`xmlparserv2.jar`	*ORACLE_DATABASE_HOME*`/lib`	*WL_HOME*`/server/lib`

3. Launch the Config Wizard (*WL_HOME*`/common/bin/config.sh` (or `.bat`).
4. Select **Extend an existing WebLogic domain**.
5. Browse and select your WebLogic Server domain.
6. Select **Extend my domain using an existing extension template**.

7. Browse and select the required template JAR (`toplink-spatial-template.jar` for Oracle Spatial, `toplink-xdb-template.jar` for Oracle XDB).
8. Complete the remaining pages of the wizard.

For information about using WebLogic Server domain templates, see *Domain Template Reference*.

2.3.9 Task 10: Start WebLogic Server and Deploy the Application

For information about deploying to WebLogic Server see *Deploying Applications to Oracle WebLogic Server* See also "Deploying Fusion Web Applications" in *Developing Fusion Web Applications with Oracle Application Development Framework*.

2.3.10 Task 11: Run the Application

For instructions for starting a deployed application from the WebLogic Administration Console, see "Start and stop a deployed Enterprise application" in *Administration Console Online Help*.

2.3.11 Task 12: Configure and Monitor Persistence Settings in WebLogic Server

In the WebLogic Server Administration Console, you can configure a persistence unit and configure JTA and non-JTA data sources of a persistence unit, as described below:

1. If you have not already done so, in the Change Center of the Administration Console, click **Lock & Edit**.
2. In the left pane of the Administration Console, select **Deployments**.
3. In the right pane, select the application or module you want to configure
4. Select **Configuration**.
5. Select **Persistence**.
6. Select the persistence unit you want to configure from the table.
7. Review and edit properties on the configuration pages. For help on any page, click the **Help** link at the top of the Administration Console.

   Properties that can be viewed include:

   - Name
   - Provider
   - Description
   - Transaction type
   - Data cache time out
   - Fetch batch size
   - Default schema name
   - Name
   - Values of persistence unit properties defined in the `persistence.xml` file, for example, `eclipselink.session-name`, `eclipselink.logging.level`, and `eclipselink.target-server`.

You can also set attributes related to the transactional and non-transactional data sources of a persistence unit, on the Data Sources configuration page.

8. To activate these changes, in the Change Center of the Administration Console, click **Activate Changes**.

For links to other help topics about working with persistence in the Administration Console, search for "Persistence" in the Table of Contents of the *Administration Console Online Help*.

2.4 Additional Resources

See the following links for more information about Oracle TopLink and Oracle WebLogic Server:

- EclipseLink Documentation Center at

  `http://wiki.eclipse.org/EclipseLink/Documentation_Center`

- Oracle WebLogic Server documentation, for example,

  Oracle Fusion Middleware Information Roadmap for Oracle WebLogic Server

2.4.1 Code Samples

See the following EclipseLink examples for related information:

- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/WebLogic_Web_Tutorial`
- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/WLS_AppScoped_DataSource`
- `http://wiki.eclipse.org/EclipseLink/Examples/Distributed`

2.4.2 Related Javadoc

For more information, see the following APIs in *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

- `org.eclipse.persistence`
- `org.eclipse.persistence.jpa.PersistenceProvider`
- `org.eclipse.persistence.services.mbean`

3

Using TopLink with GlassFish Server

This chapter describes how TopLink can be used as the persistence provider for TopLink Java Persistence API (JPA) 2.0 applications deployed to Oracle GlassFish Server.

This chapter contains the following sections:

3.1 Understanding TopLink and GlassFish Server

Oracle GlassFish Server is the reference implementation of the Java Platform, Enterprise Edition (Java EE) specification. Built using the GlassFish Server Open Source Edition, Oracle GlassFish Server delivers a flexible, lightweight and production-ready Java EE platform.

GlassFish Server is part of the Oracle Fusion Middleware application grid portfolio and is ideally suited for applications requiring lightweight infrastructure with the most up-to-date implementation of enterprise Java. GlassFish Server complements Oracle WebLogic Server, which is designed to run the broader portfolio of Oracle Fusion Middleware and large-scale enterprise applications.

3.1.1 Advantages to Using TopLink with GlassFish Server

By adding TopLink support, developers writing applications for the GlassFish platform can achieve full Java-to-data source integration compliant with the Java Persistence API (JPA) 2.0 specification. TopLink allows you to integrate Java applications with any data source, without compromising ideal application design or data integrity. In addition, TopLink gives your GlassFish platform applications the ability to store (that is, *persist*) and retrieve business domain objects using a relational database or an XML data source as a repository.

While GlassFish Server can use other persistence providers and TopLink can be used with other application servers, using GlassFish Server with TopLink provides a number of advantages:

- TopLink is included in all GlassFish Server distributions and is the default JPA provider.
- TopLink allows applications running on GlassFish Server to use Oracle Coherence caches. Coherence is a Java-based in-memory data-grid product that provides data

caching, data replication, and distributed computing services. TopLink includes features that allow deployed applications to use Coherence data caches and to incorporate TopLink Grid as an object-to-relational persistence framework. How to use this product is beyond the scope of this documentation. See *Oracle Fusion Middleware Integration Guide for Oracle TopLink with Coherence Grid* for more information.

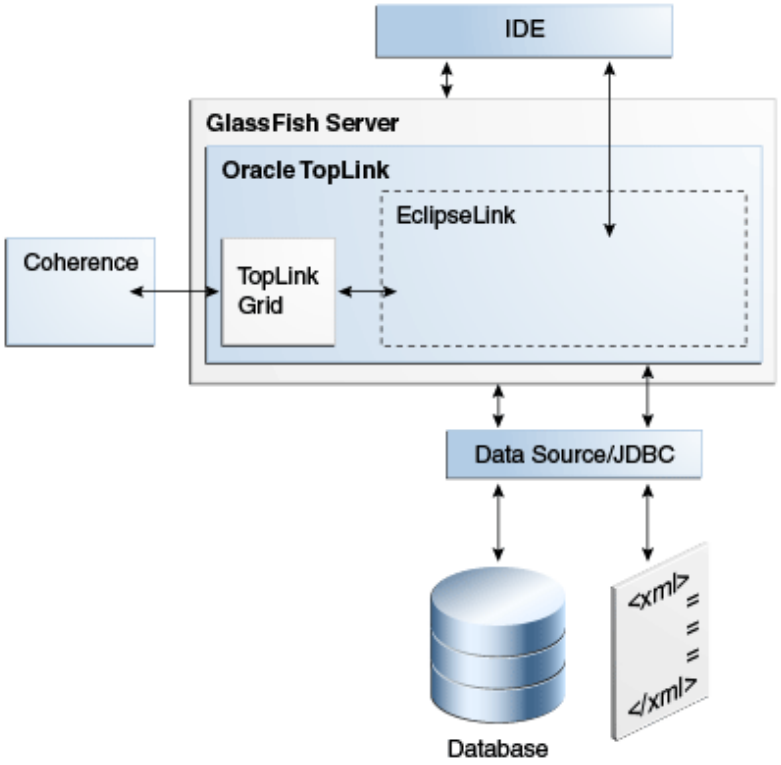
- TopLink logging integration in GlassFish Server provides a comprehensive, integrated logging infrastructure.
- GlassFish Server supports the Oracle Application Framework (ADF), an end-to-end Java EE framework, based on Struts and JavaServer Faces (JSF). ADF simplifies application development by providing infrastructure services and a visual and declarative development experience. TopLink and ADF together provide a complete Java EE application infrastructure. How to use ADF is beyond the scope of this documentation. See *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

3.1.2 Relationship of GlassFish Server and TopLink to Fusion Middleware Products

Figure 3–1 illustrates how GlassFish Server and TopLink are related to and used with other Oracle products. You can use TopLink as the persistence provider; use Oracle Coherence (through TopLink Grid integration) for data caching, data replication and distributed computing services; use GlassFish as the application server; and use the Oracle database for persisting data.

Note: Oracle Coherence and TopLink Grid are beyond the scope of this documentation. For information about Coherence, see *Oracle Coherence Developers Guide*, and follow links to other Coherence documentation. For information on TopLink Grid, see *Oracle Fusion Middleware Integration Guide for Oracle TopLink with Coherence Grid*.

Figure 3–1 Relationship of GlassFish Server and TopLink to Other Products in the Fusion Middleware Stack



3.2 What You Need to Start

This documentation is based on the following products and tools, although the principles apply to any supported database or development environment. It is assumed that the software is already installed, except where noted in later sections.

To develop and deploy TopLink applications to GlassFish Server, you need:

- GlassFish Server version 3.1.1.

  For more information and downloads, see http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html on the Oracle Technology Network.

- EclipseLink software distribution 2.3.0.

  For more information and downloads, see http://www.eclipse.org/eclipselink/ on the EclipseLink Web site.

- Any compliant JDBC database including Oracle, Oracle Express, MySQL, and so on.

  For the Oracle database, see http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html.

  For Oracle Express Edition, see http://www.oracle.com/technetwork/database/express-edition/overview/index.html.

For MySQL, see
http://www.oracle.com/us/products/mysql/index.html.

- While it is not required, you may want to use a Java EE development environment (IDE) for convenience during development. For example JDeveloper, Oracle Enterprise Pack for Eclipse (OEPE), and Oracle NetBeans all provide sophisticated Java EE development tools.

  For JDeveloper, see
  http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html.

  For OEPE, see
  http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html.

  For NetBeans, see
  http://www.oracle.com/us/products/tools/050845.html

3.3 Main Tasks

To run TopLink JPA applications in GlassFish Server, you must configure the server and coordinate certain server and application settings. These are described in the following tasks.

- Task 1: Add Object-XML (JAXB) Support to GlassFish Server (optional)
- Task 2: Set Up the Datasource
- Task 3: Create the persistence.xml File
- Task 4: Set Up GlassFish Server for JPA
- Task 5: Create the Application
- Task 6: Deploy the Application to GlassFish Server
- Task 7: Run the Application
- Task 8: Monitor the Application

3.3.1 Task 1: Add Object-XML (JAXB) Support to GlassFish Server (optional)

Oracle TopLink is included with the GlassFish distribution. You can find instructions for installing and configuring GlassFish server at this URL:

http://glassfish.java.net/docs/3.1.1/installation-guide.pdf

The TopLink modules appear as separate JAR files in the `modules` directory.

```
* \glassfish\modules
      ...
      o org.eclipse.persistence.antlr.jar
      o org.eclipse.persistence.asm.jar
      o org.eclipse.persistence.core.jar
      o org.eclipse.persistence.jpa.jar
      o org.eclipse.persistence.jpa.modelgen.jar
      o org.eclipse.persistence.oracle.jar
      ...
```

Notes:

- The `toplink-grid.jar` file, which provides support for Coherence caches, is available only if you purchase the license for Oracle Coherence. For more information on the functionality provided by the `toplink-grid.jar` file, see *Oracle Coherence Integration Guide for Oracle TopLink with Coherence Grid*.
- The `org.eclipse.persistence.oracle.jar` file is available with GlassFish and provides Oracle database-specific functionality for TopLink. This file is used only for applications running against an Oracle database.

Object-XML (also known as JAXB support, or *MOXy*) is a TopLink component that enables you to bind Java classes to XML schemas. Adding Object-XML support to GlassFish Server is optional. Many applications can run on Glassfish Server without adding Object-XML support.

Object-XML is not distributed with GlassFish 3.1.1. To get the Object-XML bundle, you must obtain it from the EclipseLink release 2.3.0 software distribution.

1. Download and open the EclipseLink release 2.3.0 software distribution.
2. Copy the Object-XML bundle `org.eclipse.persistence.moxy_2.3.0.v20110604-r9504.jar` from the distribution to the `.../glassfish/modules` folder.
3. Delete the contents of the `osgi-cache` subfolder for each of the domains in the `.../glassfish/domains` folder.
4. Restart GlassFish Server if it is already running.

Note: Beginning with release 3.1.2, Object-XML (MOXy) will be shipped with GlassFish Server.

3.3.2 Task 2: Set Up the Datasource

Configuring an Oracle database as a JDBC resource for a Java EE application involves the following tasks:

1. Integrate the JDBC Driver for Oracle Database into GlassFish Server
2. Create a JDBC Connection Pool for the Resource
3. Create the JDBC Resource

3.3.2.1 Integrate the JDBC Driver for Oracle Database into GlassFish Server

To integrate the JDBC driver, copy its JAR file into the domain and then restart the domain and instances to make the driver available.

1. Copy the JAR file for the JDBC driver into the domain's `lib` subdirectory, for example:

```
cd /home/gfuser/glassfish3
cp oracle-jdbc-drivers/ojdbc6.jar glassfish/domains/domain1/lib
```

Note that there is no need to restart GlassFish Server; the drivers are picked up dynamically.

If the application uses Oracle database-specific extensions provided by TopLink, the driver must be copied to the `lib/ext` directory. See "Oracle Database Enhancements" in the *Oracle GlassFish Server 3.1 Application Development Guide* for more information:

`http://download.oracle.com/docs/cd/E18930_01/html/821-2418/gbxjh.html#giqbi`

2. You can use the GlassFish Administration Console or the command line to restart instances in the domain to make the JDBC driver available to them.

   To use the GlassFish Server Administration Console:

   In the GlassFish Server Administration Console, open the Cluster node. Select the node for the cluster and on its General Information page, click the Instances tab. Select the instances you want to restart. For more information, see "To Start Clustered GlassFish Server Instances" in *GlassFish Administration Console Online Help*.

   To use the command line:

   Run the `restart-instance` subcommand to restart the instances. These commands assume that your instances are named `pmd-i1` and `pmd-i2`.

```
restart-instance pmd-i1
restart-instance pmd-i2
```

3.3.2.2 Create a JDBC Connection Pool for the Resource

You can create a JDBC connection pool from the GlassFish Server Administration Console or from the command line.

To use the GlassFish Server Administration Console:

In the GlassFish Server Administration Console, select the Common Tasks node, then click the **Create New JDBC Connection Pool** button in the Common Tasks page. Specify the name of the pool, the resource type, the name of the database provider, the data source and driver class names, and other details. For more information, see "To Create a JDBC Connection Pool" in *GlassFish Administration Console Online Help*.

To use the command line:

1. Use the `create-jdbc-connection-pool` subcommand to create the JDBC connection pool, specifying the database connectivity values. In this command, note the use of two backslashes (`\\`) preceding the colons in the URL property value. These backslashes cause the colons to be interpreted as part of the property value instead of as separators between property-value pairs, for example:

```
create-jdbc-connection-pool
  --datasourceclassname oracle.jdbc.pool.OracleDataSource
  --restype javax.sql.DataSource
  --property
User=coreora10g\\:Password=coreora10g\\:url=jdbc\\:oracle\\:thin\\:@asqe-db1.us
.oracle.com\\:1521\\:asqedb
    poolbvcallbackbmt
```

2. Verify connectivity to the database.

```
ping-connection-pool pool_name
```

3.3.2.3 Create the JDBC Resource

You can use the GlassFish Server Administration Console to create the JDBC resource or you can use the command line.

To use the GlassFish Server Administration Console:

In the GlassFish Server Administration Console, select the Resources node, then JDBC node, then JDBC Resources node to open the JDBC Resources page. Provide a unique JNDI resource name and associate the resource with a connection pool. For more information, see "To Create a JDBC Resource" in the *GlassFish Administration Console Online Help*.

To use the command line:

Use the `create-jdbc-resource` subcommand to create the JDBC resource, making sure to name it so that the application can discover it using JNDI lookup, for example:

```
create-jdbc-resource --connectionpoolid poolbvcallbackbmt jdbc/bvcallbackbmt
```

3.3.3 Task 3: Create the persistence.xml File

Example 3–1 illustrates a sample `persistence.xml` file which specifies the default persistence provider for TopLink, `org.eclipse.persistence.jpa.PersistenceProvider`. For more information on this file, see "Configuring Persistence Units Using persistence.xml" in the EclipseLink documentation:

`http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Configuration/JPA/persistence.xml`

If you are using the default persistence provider, you can specify additional database properties listed at "How to Use EclipseLink JPA Extensions for JDBC Connection Communication" in the EclipseLink documentation:

`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_EclipseLink_JPA_Extensions_for_JDBC_Connection_Communication`

Several of the values you enter in the file must match the values you chose when you defined the cluster, connection, and connection pool properties in GlassFish Server, as noted below:

JDBC Datasource Properties:

- **Name**: The name of the data source, which is typically the same as the JNDI name (below), for example `jdbc/bvcallbackbmt`.
- **JNDI Name**: The JNDI path to where this data source is bound. This must be the same name as the value for the `<jta-data-source>` element in `persistence.xml`, for example `jdbc/bvcallbackbmt`.
- **Database Type**: `Oracle`
- **Database Driver**: (default) Oracle's Driver (Thin XA) for Instance connections; Versions: 9.0.1 and later

Connection Properties:

- **Database Name**: The name of the database, for example, `XE` for Oracle Database Express samples.
- **Host Name**: The IP address of the database server, for example `127.0.0.1` for a locally hosted database.
- **Port**: The port number on which your database server listens for connection requests, for example, `1521`, the default for Oracle Database Express 11g.

- **Database User Name**: The database account user name used to create database connections, for example `hr` for Oracle Database Express 11g samples.
- **Password**: Your password.

Select Targets:

- **Servers / Clusters**: Select the Administration Server, managed server(s), or cluster(s) to which you want to deploy the data source. You can choose one or more.

The sample `persistence.xml` file in Example 3–1 highlights the properties defining the persistence provider, the JTA data source, and logging details. In this example, logging level is set to `FINE`. At this level, SQL code generated by EclipseLink is logged to `server.log`. For more information on these properties, see these sections:

- Section 3.3.3.1, "Specify the Persistence Provider."
- Section 3.3.3.2, "Specify an Oracle Database."
- Section 3.3.3.3, "Specify Logging."

Example 3–1 Sample persistence.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
  <persistence-unit name="pu1" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/bvcallbackbmt</jta-data-source>
    <properties>
       <property name="eclipselink.logging.level" value="FINE"/>
       <property name="eclipselink.ddl-generation"
                 value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

3.3.3.1 Specify the Persistence Provider

The persistence provider defines the implementation of JPA. It is defined in the `provider` element of the `persistence.xml` file. Persistence providers are vendor-specific. The persistence provider for Oracle TopLink is `org.eclipse.persistence.jpa.PersistenceProvider`.

3.3.3.2 Specify an Oracle Database

You specify the database connection details in the `persistence.xml` file. GlassFish Server uses the bundled Java DB (Derby) database by default, named `jdbc/__default`. To use a non-default database, such as the Oracle database, either specify a value for the `jta-data-source` element, or set the `transaction-type` element to `RESOURCE_LOCAL` and specify a value for the `non-jta-data-source` element.

If you are using the default persistence provider, `org.eclipse.persistence.jpa.PersistenceProvider`, the provider attempts to automatically detect the database type based on the connection metadata. This database type is used to issue SQL statements specific to the detected database type. You can specify the optional `eclipselink.target-database` property to guarantee that the database type is correct.

See "Specifying the Database" in the *Oracle GlassFish Server 3.1 Application Development Guide* for more information on specifying database properties in a `persistence.xml` file for GlassFish Server:

http://download.oracle.com/docs/cd/E18930_01/html/821-2418/gbwmj.html

This topic also contains a discussion about using the Java Persistence API outside the EJB container (in Java SE mode). The `eclipselink.jdbc.*` properties are specified only when using GlassFish Server in Java SE mode, for example:

```
    ...
    <property name="eclipselink.jdbc.url"
value="jdbc:oracle:thin://localhost:1521:xe;retrieveMessagesFromServerOnGetMessage
=true;create=true;"/>
    <property name="eclipselink.jdbc.user" value="APP"/>
    <property name="eclipselink.jdbc.password" value="APP"/>
    ...
```

The following are typical connection details for an Oracle database:

- **Driver**: `oracle.jdbc:OracleDriver`
- **SID** (database name): `xe`
- **Host**: `localhost`
- **Port Number**: `1521`
- **username**: name of the database user
- **password**: database password
- **Connection URL**: `jdbc:oracle:thin:@localhost:1521:xe`

For more information on these properties and other extensions for JDBC datasource connectivity, see "Using EclipseLink JPA Extensions for JDBC" and `PersistenceUnitProperties` API in the EclipseLink documentation:

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_JDBC

Also see `PersistenceUnitProperties` in *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

3.3.3.3 Specify Logging

TopLink provides a logging utility even though logging is not part of the JPA specification. Hence, the information provided by the log is TopLink JPA-specific. With TopLink, you can enable logging to view the following information:

- configuration details
- information to facilitate debugging
- the SQL that is being sent to the database

You can specify logging in the `persistence.xml` file. TopLink logging properties let you specify the level of logging and whether the log output goes to a file or standard output. Because the logging utility is based on `java.util.logging`, you can specify a logging integration to use.

The logging utility provides nine levels of logging control over the amount and detail of the log output. Use the `eclipselink.logging.level` to set the logging level, for example:

```
<property name="eclipselink.logging.level" value="FINE"/>
```

By default, the log output goes to `System.out` or to the console. To configure the output to be logged to file, set the property `eclipselink.logging.file`, for example:

```
<property name="eclipselink.logging.file" value="output.log"/>
```

TopLink's logging utility is pluggable, and several different logging integrations are supported, including `java.util.logging`. To enable `java.util.logging`, set the property `eclipselink.logging.logger`, for example:

```
<property name="eclipselink.logging.logger" value="JavaLogger"/>
```

While running inside GlassFish Server, TopLink is configured by GlassFish to use `JavaLogger` by default. The log is always redirected to the GlassFish `server.log` file. For more information, see "Setting the Logging Level" in *Oracle GlassFish Server 3.1 Application Development Guide*:

http://download.oracle.com/docs/cd/E18930_01/html/821-2418/gdpwu.html

For more information on TopLink logging and the levels of logging available in the logging utility, see "EclipseLink/Examples/JPA/Logging" in the EclipseLink documentation:

http://wiki.eclipse.org/EclipseLink/Examples/JPA/Logging

3.3.4 Task 4: Set Up GlassFish Server for JPA

The *GlassFish Server Application Development Guide* describes server-specific considerations on setting up GlassFish server to run applications that employ JPA.

http://download.oracle.com/docs/cd/E18930_01/html/821-2418/gbxjk.html

The *Application Development Guide* provides more information on these topics:

- "Specifying the Database," for information on database connection properties
- "Additional Database Properties," for information on database properties if you are using the default persistence provider
- "Configuring the Cache," for caching properties for the default persistence provider
- "Setting the Logging Level," for setting the logging properties for the default persistence provider

3.3.5 Task 5: Create the Application

To create an application that uses TopLink as its JPA provider, you may want to use a Java EE development environment for convenience during development. For example, Oracle JDeveloper, Oracle Enterprise Pack for Eclipse (OEPE) and Oracle NetBeans all provide sophisticated Java EE development tools, including support for TopLink. See "Development Tools for TopLink" in *Oracle TopLink Concepts*.

For guidance in writing your application see these topics from the "Using the Java Persistence API" chapter in *Oracle GlassFish Server 3.1 Application Development Guide*:

http://download.oracle.com/docs/cd/E18930_01/html/821-2418/gbxjk.html

3.3.6 Task 6: Deploy the Application to GlassFish Server

For information about deploying to GlassFish Server see "Deploy Applications or Modules," "To Deploy an Enterprise Application," and "To Deploy a Web Application" in the *GlassFish Server Administration Console Online Help*. See also *Oracle GlassFish Server 3.1 Application Deployment Guide*:

http://download.oracle.com/docs/cd/E18930_01/html/821-2417/index.html

3.3.7 Task 7: Run the Application

For instructions for starting a deployed application from the GlassFish Administration Console, see "Application Client Launch" and "To Launch an Application" in *GlassFish Server Administration Console Online Help*.

3.3.8 Task 8: Monitor the Application

GlassFish Server provides a monitoring service to track the health and performance of an application. For information on monitoring an application from the console, see the "Monitoring" and "Monitoring Data" topics in *GlassFish Server Administration Console Online Help*. For information on monitoring the application from the command line, see "Administering the Monitoring Service" in *Oracle GlassFish Server 3.1 Application Deployment Guide*:

http://download.oracle.com/docs/cd/E18930_01/html/821-2416/ablur.html

3.4 Additional Resources

See the following links for more information about Oracle TopLink and Oracle GlassFish Server.

- EclipseLink documentation

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

- Oracle GlassFish Server 3.1 Application Deployment Guide

  http://download.oracle.com/docs/cd/E18930_01/html/821-2417/index.html

- Oracle GlassFish Server Application Development Guide

  http://download.oracle.com/docs/cd/E18930_01/html/821-2418/gbxjk.html

- Oracle GlassFish Server 3.1 - 3.1.1 Documentation Library

  http://download.oracle.com/docs/cd/E18930_01/index.html

4

Using Multiple Databases with a Composite Persistence Unit

With TopLink, you can expose multiple persistence units (each with unique sets of entity types) as a single persistence context by using a *composite persistence unit*. Individual persistence units that are part of a composite persistence unit are called *composite member persistent units*.

This chapter includes the following sections:

4.1 Understanding the Composite Persistence Unit

With a composite persistence unit, you can:

- Map relationships among any of the entities in the composite persistence unit
- Access entities stored in multiple databases and different data sources
- Easily perform queries and transactions across the complete set of entities

Example 4–1 shows how you can persist data from a single context into two different databases:

Example 4–1 Using Multiple Databases

```
em.persist(new A(..));
em.persist(new B(..));
// You can insert A into database1 and insert B into database2.
// The two databases can be from different vendors.

em.flush();
```

Figure 4–1 Simple Composite Persistence Unit

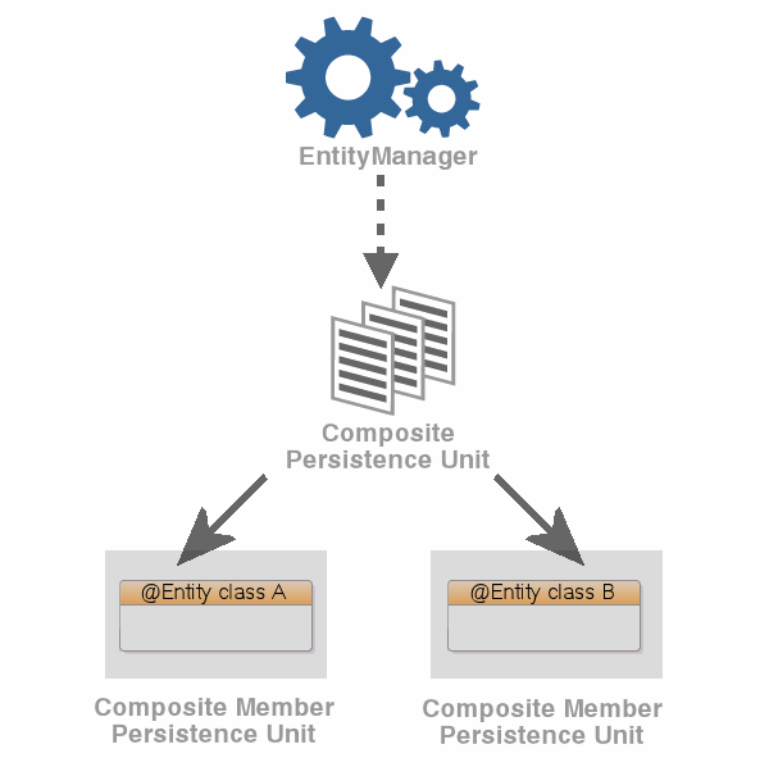


Figure 4–1 illustrates a simple composite persistence unit. The `EntityManager` (passing them to `Persistence.createEntityManagerFactory` method) instantiates the composite persistence unit, which contains two composite member persistence units:

- Class **A** is mapped by a persistence unit named **memberPu1** located in `member1.jar`.
- Class **B** is mapped by a persistence unit named **memberPu2** located in `member2.jar`.

4.1.1 Composite Persistence Unit Requirements

When using composite persistence units, be aware of the following requirements:

- The name of each composite member persistence unit must be unique within the composite.
- The `transaction-type` and other properties that correspond to the entire persistence unit (such as target server, logging, transactions, and so on) should be defined in the composite persistence unit. If not, the application uses the defaults."
- All composite members persistence units should use the same transaction-type as the composite persistence unit.

4.2 Main Tasks

This section includes the following tasks:

- Task 1: Configure the Composite Persistence Unit
- Task 2: Use Composite Persistence Units

- Task 3: Deploy Composite Persistence Units

4.2.1 Task 1: Configure the Composite Persistence Unit

Because the composite persistence unit is a regular persistence element, it requires a `persistence.xml` file. Example 4–2 illustrates a sample `persistence.xml` file:

Example 4–2 The persistence.xml File for a Composite Persistence Unit

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd"
version="1.0">
    <persistence-unit name="compositePu" transaction-type="JTA">
        <provider>
            org.eclipse.persistence.jpa.PersistenceProvider
        </provider>

        <jar-file>member1.jar</jar-file>
        <jar-file>member2.jar</jar-file>
        <properties>
            <property name="eclipselink.composite-unit" value="true"/>
            <property name="eclipselink.target-server" value="WebLogic_10"/>
        </properties>
    </persistence-unit>
</persistence>
```

Use the `<property name="eclipselink.composite-unit" value="true"/>` property to identify persistence unit as a composite.

Use the `<jar-file>` element to specify the JAR files containing the composite member persistent units. The composite persistence unit will contain all the persistence units found in the JAR files specified.

4.2.2 Task 2: Use Composite Persistence Units

You can use a composite persistence unit as you would any other persistence unit; the `EntityManager` could be injected, as shown here:

```
@PersistenceUnit(unitName="compositePu")
EntityManagerFactory entityManagerFactory;

@PersistenceUnit(unitName="compositePu")
EntityManager entityManager;
```

Or create it manually:

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("compositePu", properties);
```

4.2.3 Task 3: Deploy Composite Persistence Units

To deploy multiple persistence units, deploy all of the JARs (the composite and its members) on he same class loader.

- When deploying to Oracle WebLogic Server, package the JARs in an EAR file.
- When running as a standalone application, add the JARs to the class path.

See Section 4.1.1, "Composite Persistence Unit Requirements" for important information and limitations.

4.3 Additional Resources

See `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Composite_Persistence_Units` in the EclipseLink documentation for additional information on composite persistence units including:

- Limitations of composite persistence units.
- Configuring composite member persistence units that contain dependencies to each other.
- All persistence unit properties used by composite persistence units and composite member persistence units
- How to pass persistence unit properties to composite member persistence units with the `Persistence.createEntityManagerFactory` method, while creating a composite persistence unit EntityManagerFactory
- All entity manager properties used by composite persistence unit and composite member persistence units
- How to pass entity manager properties to composite member persistence units with the `emf.createEntityManager` method for composite persistence unit `EntityManagerFactory`.

4.3.1 Javadoc

For more information, see the following APIs in *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

- `PersistenceUnitProperties` class
- `Persistence.createEntityManger` class
- `EntityManagerFactory` interface

5

Scaling TopLink Applications in Clusters

This chapter provides instructions for configuring TopLink applications to ensure scalability in clustered application server environments. The instructions are generic and can be applied to any clustered application server environment; however, additional content is provided for WebLogic server and GlassFish server. Consult your vendor's documentation as required.

This chapter contains the following sections:

5.1 Understanding Scaling TopLink Applications in Clusters

TopLink applications that are deployed to clustered application server environments benefit from cluster scalability, load balancing, and failover. These capabilities ensure that TopLink applications are highly available and can scale as application demand increases. TopLink applications are deployed the same way in clustered server environments as they are in standalone server environments. However, TopLink applications must consider cache consistency in clustered environments.

TopLink utilizes a shared (L2) object cache that avoids database access for objects and their relationships. The cache is enabled by default and enhances application performance. In clustered environments, caching can result in consistency issues (such as stale data) as changes made on one server are not reflected on objects cached in other servers. Cache consistency is only problematic for objects that are frequently updated. Read-only objects are not affected by cache consistency. See the EclipseLink documentation for detailed information on caching:

`http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Caching/Caching_Overview`

Various options are available for addressing cache consistency:

- Use distributed caching. TopLink includes an integration with Oracle Coherence that addresses many cache consistency issues that result from operating in a distributed environment. The integration is beyond the scope of this documentation. See *Oracle Coherence Integration Guide for Oracle TopLink with Coherence Grid* for additional details.
- Use cache coordination to broadcast changes between the servers in the cluster to update or invalidate changed objects.

- Use optimistic locking to prevent updates to stale objects and trigger the objects to be invalidated in the cache.
- Use object/query refreshing when fresh data is required
- Disable the shared cache or only cache read-only objects

5.2 Main Tasks

The tasks in this section provide general instructions for ensuring that a TopLink application can scale in an application server cluster environment. These tasks must be completed prior to deploying an application.

This section contains the following tasks:

- Task 1: Configure Cache Consistency
- Task 2: Ensure TopLink is Enabled
- Task 3: Ensure All Application Servers are Part of the Cluster

5.2.1 Task 1: Configure Cache Consistency

This task includes different configuration options that mitigate the possibility or chance that an application might use stale data when deployed to an application server cluster environment. The cache coordination option is specifically designed for clustered applications; however, evaluate all the options and use them together (if applicable) to create a solution that results in the best application performance. Properly configuring a cache can, in some cases, eliminate the need to use cache coordination. For additional details on these options, see:

`http://wiki.eclipse.org/Introduction_to_Cache_%28ELUG%29#Handling_Stale_Data`

The following topics are included in this section:

- Section 5.2.1.1, "Disabling the Shared Cache"
- Section 5.2.1.2, "Refreshing the Cache"
- Section 5.2.1.3, "Setting Cache Expiration"
- Section 5.2.1.4, "Setting Optimistic Locking"
- Section 5.2.1.5, "Using Cache Coordination"

> **Note:** Oracle provides a TopLink and Coherence integration that allows TopLink to use Coherence as the L2 cache. The integration is beyond the scope of this documentation. See *Oracle Coherence Integration Guide for Oracle TopLink with Coherence Grid* for additional details.

5.2.1.1 Disabling the Shared Cache

Cache consistency can be avoided by disabling the shared cache if an application does not require shared caching. To disable the shared cache for all objects, use the `<shared-cache-mode>` element in the `persistence.xml` file. For example:

```
<shared-cache-mode>NONE</shared-cache-mode>
```

To selectively enable or disable the shared cache, use the `shared` attribute of the `@Cache` annotation when defining an entity. For example:

```
...
@Entity
@Cache(shared=false)
public class Employee {
  ...
}
```

5.2.1.2 Refreshing the Cache

Refreshing a cache reloads the cache from the database to ensure that an application is using current data. This section describes different ways to refresh a cache.

The `@cache` annotation provides the `alwaysRefresh` and `refreshOnlyIfNewer` attributes which force all queries that go to the database to refresh the cache:

```
...
@Entity
@Cache(
   alwaysRefresh=true,
   refreshOnlyIfNewer=true)
public class Employee {
  ...
}
```

The `org.eclipse.persistence.jpa.JpaCache` interface includes several methods that remove stale objects if the cache is out of date:

- The `evictAll` method invalidates all of the objects in the cache. For example:

  ```
  em.getEntityManagerFactory().getCache().evictAll();
  ```

- Use the `evict` method to invalidate specific classes.

  ```
  em.getEntityManagerFactory().getCache().evict(MyClass);
  ```

- The `clear` method also refreshes a cache; however, clearing the cache can cause object identity issues if any of the cached objects are in use. Use this method only if the application knows that it no longer has references to objects held in the cache.

The preceding methods are passive and only refresh objects the next time the cache is accessed. To actively refresh an object, use the `EntityManager.refresh` method. The method refreshes a single object at a time.

Any of the following APIs also refresh a cache:

- `Session.refreshObject`
- `DatabaseSession` and `UnitOfWork`: `refreshAndLockObject` methods
- `ObjectLevelReadQuery`: `refreshIdentityMapResult` and `refreshRemoteIdentityMapResult` methods

The `ClassDescriptor` class also provides methods that refresh a cache:

- `setShouldAlwaysRefreshCache`
- `setShouldAlwaysRefreshCacheOnRemote`
- `setShouldDisableCacheHits`
- `setShouldDisableCacheHitsOnRemote`
- `setShouldOnlyRefreshCacheIfNewerVersion`

Use these methods in a descriptor amendment method. For example:

```
public void addToDescriptor(ClassDescriptor descriptor) {
    descriptor.setShouldRefreshCacheOnRemote(true);
    descriptor.setShouldDisableCacheHitsOnRemote(true);
}
```

Lastly, a query hint triggers a query to refresh the cache. For example:

```
Query query = em.createQuery("Select e from Employee e");
query.setHint("javax.persistence.cache.storeMode", "REFRESH");
```

5.2.1.3 Setting Cache Expiration

Cache expiration makes a cached object instance invalid after a specified amount of time. Any attempt to use the object causes the most up-to-date version of the object to be reloaded from the data source. Expiration can help ensure that an application is always using the most recent data. This section describes different ways to set expiration.

The `@cache` annotation provides the `expiry` and `expiryTimeOfDay` attributes which remove cache instances after a specific amount of time. The `expiry` attribute is entered in milliseconds. The default value if no value is specified is `-1` which indicates that expiry is disabled. The `expiryTimeOfDay` attribute is an instance of the `org.eclipse.persistence.annotations.TimeOfDay` interface. The following example sets the object to expire after 5 minutes:

```
...
@Entity
@Cache(expiry=300000)
public class Employee {
  ...
}
```

At the descriptor level, use the `ClassDescriptor.setCacheInvalidationPolicy` method to set a `CacheInvalidationPolicy` instance. The following invalidation policies are available:

- `DailyCacheInvalidationPolicy`: the object is automatically flagged as invalid at a specified time of day.
- `NoExpiryCacheInvalidationPolicy`: the object can only be flagged as invalid by explicitly calling `IdentityMapAccessor.invalidateObject` method.
- `TimeToLiveCacheInvalidationPolicy`: the object is automatically flagged as invalid after a specified time period has elapsed since the object was read.

5.2.1.4 Setting Optimistic Locking

Optimistic locking prevents one user from writing over another user's work. Locking is important when multiple servers or multiple applications access the same data and is relevant in both single-server and multiple-server environments. In a multiple-server environment, locking is still required if an application uses cache refreshing or cache coordination. This section describes different ways to set optimistic locking.

The `@OptimisticLocking` annotation specifies the type of optimistic locking to use when updating or deleting entities. Optimistic locking is supported on an `@Entity` or `@MappedSuperclass` annotation. The following attributes are available:

- `ALL_COLUMNS`: This policy compares every field in the table in the `WHERE` clause when doing an update or a delete.
- `CHANGED_COLUMNS`: This policy compares only the changed fields in the `WHERE` clause when doing an update. A delete operation will only compare the primary key.
- `SELECTED_COLUMNS`: This policy compares selected fields in the `WHERE` clause when doing an update or a delete. The fields specified must be mapped and not be primary keys.
- `VERSION_COLUMN`: (default) This policy allows a single version number to be used for optimistic locking. The version field must be mapped and not be the primary key. To automatically force a version field update on a parent object when its privately owned child object's version field changes, use the `cascaded` method set to `true`. The method is set to `false` by default.

At the descriptor level, configure optimistic locking by using the `ClassDescriptor.setOptimisticLockingPolicy` method to set an optimistic `FieldsLockingPolicy` instance. As with the annotation, the following policies are included:

- `AllFieldsLockingPolicy`: This policy compares every field in the table in the `WHERE` clause when doing an update or a delete.
- `ChangedFieldsLockingPolicy`: This policy compares only the changed fields in the `WHERE` clause when doing an update. A delete operation will only compare the primary key.
- `SelectedFieldsLockingPolicy`: This policy compares selected fields in the `WHERE` clause when doing an update or a delete. The fields specified must be mapped and not be primary keys.
- `VersionLockingPolicy`: This policy is used to allow a single version number to be used for optimistic locking. To automatically force a version field update on a parent object when its privately owned child object's version field changes, use the `VersionLockingPolicy.setIsCascaded` method set to `true`.
- `TimestampLockingPolicy`: This policy is used to allow a single version timestamp to be used for optimistic locking.

5.2.1.5 Using Cache Coordination

Cache coordination synchronizes changes among distributed sessions. Cache coordination is most useful in application server clusters where the need to maintain consistent data for all applications can be challenging. Moreover, cache consistency becomes increasingly more difficult as the number of servers within an environment increases.

Cache coordination works by broadcasting notifications of transactional object changes among sessions (`ServerSession` or persistence unit) in the cluster. Cache coordination is most useful for application that are primarily read-based and when changes are performed by the same application operating with multiple, distributed sessions.

Cache coordination significantly minimizes stale data, but does not completely eliminate the possibility that stale data might occur. In addition, cache coordination reduces the number of optimistic lock exceptions encountered in a distributed architecture, and decreases the number of failed or repeated transactions in an application. However, cache coordination in no way eliminates the need for an

effective locking policy. To ensure the most current data, use cache coordination with optimistic or pessimistic locking; optimistic locking is preferred.

Cache coordination is supported over RMI and JMS and can be configured either declaratively by using persistence properties in a `persistence.xml` file or by using the cache coordination API. System properties that match the persistence properties are available as well.

For additional details on cache coordination see:

http://wiki.eclipse.org/Introduction_to_Cache_%28ELUG%29#Cache_Coordination_2

Configuring JMS Cache Coordination Using Persistence Properties

The following example demonstrates how to configure cache coordination in the `persistence.xml` file and uses JMS for broadcast notification. For JMS, provide a JMS topic JNDI name and topic connection factory JNDI name in addition to the protocol. The JMS topic should not be JTA enabled and should not have persistent messages.

```
<property name="eclipselink.cache.coordination.protocol" value="jms" />
<property name="eclipselink.cache.coordination.jms.topic"
   value="jms/EmployeeTopic" />
<property name="eclipselink.cache.coordination.jms.factory"
   value="jms/EmployeeTopicConnectionFactory" />
```

Applications that run in a cluster generally do not require a URL as the topic is enough to locate and use the resource. For applications that run outside the cluster, a URL is required. The following example is a URL for a WebLogic server cluster:

```
<property name="eclipselink.cache.coordination.jms.host"
   value="t3://myserver:7001/" />
```

A user name and password for accessing the servers can also be set if required. For example:

```
<property name="eclipselink.cache.coordination.jndi.user" value="user" />
<property name="eclipselink.cache.coordination.jndi.password" value="password" />
```

Configuring RMI Cache Coordination Using Persistence Properties

The following example demonstrates how to configure cache coordination in the `persistence.xml` file and uses RMI for broadcast notification.

```
<property name="eclipselink.cache.coordination.protocol" value="rmi" />
```

Applications that run in a cluster generally do not require a URL because JNDI is replicated and each server can look up each others listener. If an application runs outside of a cluster, or if JNDI is not replicated, then each server must provide its URL. This could be done through the `persistence.xml` file; however, different `persistence.xml` files (thus JAR or EAR) for each server is required, which is normally not desirable. A second option is to set the URL programmatically using the cache coordination API. See "Configuring Cache Coordination Using the Cache Coordination API" on page 5-7. The final option is to set the URL as a system property on each application server. The following example sets the URL for a WebLogic server cluster using a system property:

```
-Declipselink.cache.coordination.jms.host=t3://myserver:7001/
```

A user name and password for accessing the servers can also be set if required; for example:

```
<property name="eclipselink.cache.coordination.jndi.user" value="user" />
<property name="eclipselink.cache.coordination.jndi.password" value="password" />
```

RMI cache coordination can use either asynchronous or synchronous broadcasting; asynchronous is the default. Synchronous broadcasting ensures that all of the servers are updated before the request returns. The following example configures synchronous broadcasting.

```
<property name="eclipselink.cache.coordination.propagate-asynchronously"
   value="false" />
```

If multiple applications on the same server or network use cache coordination a separate channel can be used for each application. For example:

```
<property name="eclipselink.cache.coordination.channel" value="EmployeeChannel" />
```

Lastly, if required, change the default RMI multicast socket address that allows servers to find each other. The following example explicitly configures the multicast settings:

```
<property name="eclipselink.cache.coordination.rmi.announcement-delay"
   value="1000" />
<property name="eclipselink.cache.coordination.rmi.multicast-group"
   value="239.192.0.0" />
<property name="eclipselink.cache.coordination.rmi.multicast-group.port"
   value="3121" />
<property name="eclipselink.cache.coordination.packet-time-to-live" value="2" />
```

Configuring Cache Coordination Using the Cache Coordination API

Use the `CommandManager` interface to programmatically configure cache coordination for a session. The following example configures RMI cache configuration:

```
Session.getCommandManager().setShouldPropagateAsynchronously(boolean)

Session.getCommandManager().getDiscoveryManager().
    setAnnouncementDelay()
    setMulticastGroupAddress()
    setMulticastPort()
    setPacketTimeToLive()

Session.getCommandManager().getTransportManager().
    setEncryptedPassword()
    setInitialContextFactoryName()
    setLocalContextProperties(Hashtable)
    setNamingServiceType() //passing in one of:
        TransportManager.JNDI_NAMING_SERVICE
        TransportManager.REGISTRY_NAMING_SERVICE
    setPassword()
    setRemoteContextProperties(Hashtable)
    setShouldRemoveConnectionOnError()
    setUserName()
```

Setting Cache Synchronization

Cache synchronization determines how objects changes are broadcast among session members. The following synchronization modes are available:

- SEND_OBJECT_CHANGES: (Default) This option sends a list of changed objects including data about the changes. This data is merged into the receiving cache.
- INVALIDATE_CHANGED_OBJECTS: This option sends a list of the identities of the objects that have changed. The receiving cache invalidates the objects rather than changing any of the data.
- SEND_NEW_OBJECTS_WITH_CHANGES: This option is the same as the SEND_OBJECT_CHANGES option except it also includes any newly created objects from the transaction.
- NONE: This option does no cache coordination.

The @cache annotation coordinationType attribute is used to specify the synchronization mode. For example:

```
...
@Entity
@Cache(CacheCoordinationType.SEND_NEW_OBJECTS_CHANGES)
public class Employee {
  ...
}
```

The ObjectChangeSet.setCacheSynchronizationType method can also be used to set the synchronization mode. For example

```
setCacheSynchronizationType() // passing in one of:
    ClassDescriptor.DO_NOT_SEND_CHANGES
    ClassDescriptor.INVALIDATE_CHANGED_OBJECTS
    ClassDescriptor.SEND_NEW_OBJECTS_WITH_CHANGES
    ClassDescriptor.SEND_OBJECT_CHANGES
```

5.2.2 Task 2: Ensure TopLink is Enabled

Ensure the TopLink JAR files are included on the classpath of each application server in the cluster to which the TopLink application is deployed and configure TopLink as the persistence provider. See Chapter 2, "Using TopLink with WebLogic Server," and Chapter 3, "Using TopLink with GlassFish Server," for detailed instructions on setting up TopLink with WebLogic server and GlassFish, respectively.

5.2.3 Task 3: Ensure All Application Servers are Part of the Cluster

Configure an application server cluster that includes each application server that hosts the TopLink application:

- For WLS clustering see *Using Clusters for Oracle WebLogic Server*.
- For GlassFish clustering, see:

  http://download.oracle.com/docs/cd/E18930_01/html/821-2426/index.html

5.3 Additional Resources

The following additional resources are available:

- Section 5.3.1, "Code Samples"
- Section 5.3.2, "Related JavaDoc"

5.3.1 Code Samples

http://wiki.eclipse.org/EclipseLink/Examples/JPA/CacheCoordination

http://wiki.eclipse.org/EclipseLink/Examples/JPA/Caching

5.3.2 Related JavaDoc

For more information, see the following APIs in *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

- `org.eclipse.persistence.annotations.OptimisticLocking`
- `org.eclipse.persistence.annotations.Cache`
- `org.eclipse.persistence.descriptors.ClassDescriptor`
- `org.eclipse.persistence.sessions.coordination`

6 Providing Software as a Service

This chapter provides instructions for creating shared TopLink applications that run in software as a service (SaaS) environments.

This chapter includes the following sections:

- Section 6.1, "Understanding Oracle TopLink as a SaaS"
- Section 6.2, "Making JPA Entities Extensible"
- Section 6.3, "Making JAXB Beans Extensible"
- Section 6.4, "Using Single-Table Multi-Tenancy"
- Section 6.5, "Using an External Metadata Source"

6.1 Understanding Oracle TopLink as a SaaS

The Oracle Platform for SaaS includes TopLink, part of Oracle Fusion Middleware. This allows you to build, deploy, and manage SaaS applications. With TopLink you can manage persistence in a cloud-enabled applications and services. Developing more-flexible SaaS solutions that address multi-tenancy and extensibility while still maintaining high performance and scalability makes the persistence layer of these applications a critical component.

TopLink supports providing software as a service by supporting extensibility, multi-tenancy, and the ability to use external metadata sources, as explained in the following sections:

- Section 6.2, "Making JPA Entities Extensible"
- Section 6.3, "Making JAXB Beans Extensible"
- Section 6.4, "Using Single-Table Multi-Tenancy"
- Section 6.5, "Using an External Metadata Source"

6.2 Making JPA Entities Extensible

Use the `@VirtualAccessMethods` annotation to specify that an entity is extensible. By using virtual properties in an extensible entity, you can specify mappings external to the entity. This allows you to modify the mappings without modifying the entity source file and without redeploying the entity's persistence unit.

Extensible entities are useful in a multi-tenant (or SaaS) environment where a shared, generic application can be used by multiple clients (tenants). Tenants have private access to their own data, and to data shared with other tenants.

Using extensible entities, you can:

- Build an application where some mappings are common to all users and some mappings are user-specific.
- Add mappings to an application after it is made available to a customer (even post-deployment).
- Use the same `EntityManagerFactory` to work with data after mappings have changed.
- Provide an additional source of metadata to be used by an application.

6.2.1 Main Tasks

To create and support an extensible JPA entity:

- Task 1: Configure the Entity
- Task 2: Design the Schema
- Task 3: Provide Additional Mappings
- Task 4: Configure Persistence Properties and the Data Repository

6.2.1.1 Task 1: Configure the Entity

Configuring the entity consists of annotating the entity class with `@VirtualAccessMethods`, adding `get` and `set` methods for the property values, and adding a data structure to store the extended attributes and values.

6.2.1.1.1 Annotate the Entity Class with @VirtualAccessMethods Annotate the entity with `@VirtualAccessMethods` to specify that it is extensible and to define virtual properties.

Table 6–1 describes the attributes available to the `@VirtualAccessMethods` annotation.

Table 6–1 Attributes for the @VirtualAccessMethods Annotation

Attribute	Description
`get`	The name of the getter method to use for the virtual property. This method must take a single `java.lang.String` parameter and return a `java.lang.Object`. Default: `get` Required? No
set	The name of the setter method to use for the virtual property. This method must take a `java.lang.String` and a `java.lang.Object` parameter and return a `java.lang.Object` parameter. Default: `set` Required? No

6.2.1.1.2 Add get and set Methods to the Entity Add `get(String)` and `set(String, Object)` methods to the entity. The `get()` method returns a value by property name and the `set()` method stores a value by property name. The default names for these methods are `get` and `set`, and they can be overridden with the `@VirtualAccessMethods` annotation.

EclipseLink weaves these methods if weaving is enabled, which provides support for lazy loading, change tracking, fetch groups, and internal optimizations. You must use

the `get(String)` and `set(String, Object)` signatures, or else weaving will not work.

> **Note:** Weaving is not supported when using virtual access methods with OneToOne mappings. If attempted, an exception will be thrown.

6.2.1.1.3 Add a Data Structure Add a data structure to store the extended attributes and values, that is, the virtual mappings. These can then be mapped to the database. See Section 6.2.1.3, "Task 3: Provide Additional Mappings."

A common way to store the virtual mappings is in a `Map` (as shown in Example 6–1), but you can also use other strategies. For example you could store the virtual mappings in a directory system.

When using field-based access, annotate the data structure with `@Transient` so the structure cannot be used for another mapping. When using property-based access, `@Transient` is unnecessary.

Example 6–1 illustrates an entity class that uses property access.

Example 6–1 Entity Class that Uses Property Access

```
@Entity
@VirtualAccessMethods
public class Customer{

    @Id
    private int id;
    ...

    @Transient
    private Map<String, Object> extensions;

    public <T> T get(String name) {
        return (T) extentions.get(name);
    }

    public Object set(String name, Object value) {
        return extensions.put(name, value);
    }
```

6.2.1.1.4 Use XML As an alternative to, or in addition to, using `@VirtualAccessMethods`, you can use the `<access>` and `<access-methods>` elements, for example:

```
<access>VIRTUAL</access>
<access-methods set-method="get" get-method="set"/>
```

6.2.1.2 Task 2: Design the Schema

Provide database tables with extra columns for storing flexible mapping data. For example, the following `Customer` table includes two predefined columns, `ID` and `NAME`, and three flexible columns, `FLEX_COL1`, `FLEX_COL2`, `FLEX_COL3`:

- CUSTOMER
  - INTEGER ID
  - VARCHAR NAME

- VARCHAR FLEX_COL1
- VARCHAR FLEX_COL2
- VARCHAR FLEX_COL3

You can then specify which of the flex columns should be used to persist an extended attribute, as described in "Task 3: Provide Additional Mappings".

6.2.1.3 Task 3: Provide Additional Mappings

To provide additional mappings, add the mappings with the `column` and `access-methods` attributes to the `eclipselink-orm.xml` file, for example:

```
<basic name="idNumber" attribute-type="String">
  <column name="FLEX_COL1"/>
  <access-methods get-method="get" set-method="set"/>
</basic>
```

6.2.1.4 Task 4: Configure Persistence Properties and the Data Repository

Configure persistence unit properties to indicate that the application should retrieve the flexible mappings from the `eclipselink-orm.xml` file. You can set persistence unit properties using `persistence.xml` or by setting properties on the `EntityManagerFactory`, as described in the following sections.

For more information about external mappings, see "External Mappings" in the EclipseLink documentation.

http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/External_Mappings

6.2.1.4.1 Configure persistence.xml In `persistence.xml` file, use the `eclipselink.metadata-source` property to use the default `eclipselink-orm.xml` file. Use the `eclipselink.metadata-source.xml.url` property to use a different file at the specified location, for example:

```
<property name="eclipselink.metadata-source" value="XML"/>
<property name="eclipselink.metadata-source.xml.url" value="foo://bar"/>
```

6.2.1.4.2 Configure the EntityManagerFactory and the Metadata Repository Extensions are added at bootstrap time through access to a metadata repository. The metadata repository is accessed through a class that provides methods to retrieve the metadata it holds. The current release includes a metadata repository implementation that supports XML repositories.

Specify the class to use and any configuration information for the metadata repository through persistence unit properties. The entity manager factory integrates additional mapping information from the metadata repository into the metadata it uses to bootstrap.

You can provide your own implementation of the class to access the metadata repository. Each metadata repository access class must specify an individual set of properties to use to connect to the repository.

You can subclass either of the following classes:

- `org.eclipse.persistence.internal.jpa.extensions.MetadataRepository`
- `org.eclipse.persistence.internal.jpa.extensions.XMLMetadataRepository`

In the following example, the properties that begin with `com.foo` are defined by the developer.

```
<property name="eclipselink.metadata-source" value="com.foo.MetadataRepository"/>
<property name="com.foo.MetadataRepository.location" value="foo://bar"/>
<property name="com.foo.MetadataRepository.extra-data" value="foo-bar"/>
```

6.2.1.4.3 Refresh the Metadata Repository If you change the metadata and you want an `EntityManager` based on the new metadata, you must call `refreshMetadata()` on the `EntityManagerFactory` to refresh the data. The next `EntityManager` will be based on the new metadata.

The `refreshMetadata` method takes a `Map` of properties, and that map of properties can be used to override the properties previously defined for the `metadata-source`.

6.2.2 Code Examples

Example 6–2 illustrates the following:

- Field access is used for non-extension fields.
- Virtual access is used for extension fields, using defaults (`get(String)` and `set(String, Object)`).
- The `get(String)` and `set(String, Object)` methods will be woven, even if no mappings use them, because of the presence of `@VirtualAccessMethods`.
- Extensions are mapped in a portable way by specifying `@Transient`.

Example 6–2 Virtual Access Using Default get and set Method Names

```
@Entity
@VirtualAccessMethods
public class Address {

    @Id
    private int id;

    @Transient
    private Map<String, Object> extensions;

    public int getId(){
        return id;
    }

    public <T> T get(String name) {
        return (T) extentions.get(name);
    }

    public Object set(String name, Object value) {
        return extensions.put(name, value);
    }

   ...
```

Example 6–3 illustrates the following:

- Field access is used for non-extension fields.
- The `@VirtualAccessMethods` annotation overrides methods to be used for getting and setting.

- The `get(String)` and `set(String, Object)` methods will be woven, even if no mappings use them, because of the presence of `@VirtualAccessMethods`.
- Extensions are mapped in a portable way by specifying `@Transient`.
- The XML for extended mapping indicates which `get()` and `set()` method to use.

Example 6–3 Overriding Get and Set Methods

```
@Entity
@VirtualAccessMethods(get="getExtension", set="setExtension")
public class Address {

    @Id
    private int id;

    @Transient
    private Map<String, Object> extensions;

    public int getId(){
        return id;
    }

    public <T> T getExtension(String name) {
        return (T) extensions.get(name);
    }

    public Object setExtension(String name, Object value) {
        return extensions.put(name, value);
    }

    ...

    <basic name="name" attribute-type="String">
      <column name="FLEX_1"/>
      <access-methods get-method="getExtension" set-method="setExtension"/>
    </basic>
```

Example 6–4 illustrates the following:

- Property access is used for non-extension fields.
- Virtual access is used for extension fields, using defaults (`get(String)` and `set(String, Object)`).
- The extensions are mapped in a portable way. `@Transient` is not required, because property access is used.
- The `get(String)` and `set(String, Object)` methods will be woven, even if no mappings use them, because of the presence of `@VirtualAccessMethods`.

Example 6–4 Using Property Access

```
@Entity
@VirtualAccessMethods
public class Address {

    private int id;

    private Map<String, Object> extensions;
```

```
    @Id
    public int getId(){
        return id;
    }

    public <T> T get(String name) {
        return (T) extensions.get(name);
    }

    public Object set(String name, Object value) {
        return extensions.put(name, value);
    }

...
```

6.3 Making JAXB Beans Extensible

Use the `@XmlVirtualAccessMethods` annotation to specify that a JAXB bean is extensible. By using virtual properties in an extensible bean, you can specify mappings external to the bean. This allows you to modify the mappings without modifying the bean source file and without redeploying the bean's persistence unit.

In a multi-tenant (or SaaS) architecture, a single application runs on a server, serving multiple client organizations (tenants). Good multi-tenant applications allow per-tenant customizations. When these customizations are made to data, it can be difficult for the binding layer to handle them. JAXB is designed to work with domain models that have real fields and properties. EclipseLink Object-XML 2.3 (also known as *MOXy*) introduces the concept of virtual properties which can easily handle this use case. Virtual properties are defined by the Object-XML metadata file, and provide a way to extend a class without modifying the source.

This section has the following subsections:

- Section 6.3.1, "Main Steps"
- Section 6.3.2, "Code Examples"

6.3.1 Main Steps

To create and support an extensible JAXB bean:

- Task 1: Configure the Bean
- Task 2: Provide Additional Mappings

6.3.1.1 Task 1: Configure the Bean

Configuring the bean consists of annotating the bean class with the `@XmlVirtualAccessMethods`, adding `get` and `set` methods for the property values, and adding a data structure to store the extended attributes and values.

6.3.1.1.1 Annotate the Bean Class with @Xml VirtualAccessMethods Annotate the bean with `@XmlVirtualAccessMethods` to specify that it is extensible and to define virtual properties.

Table 6–2 describes the attributes available to the `@XmlVirtualAccessMethods` annotation.

Table 6–2 Attributes for the @XmlVirtualAccessMethods Annotation

Attribute	Description
`get`	The name of the getter method to use for the virtual property. This method must take a single `java.lang.String` parameter and return a `java.lang.Object`. Default: `get` Required? No
`set`	The name of the setter method to use for the virtual property. This method must take a `java.lang.String` and a `java.lang.Object` parameter and return a `java.lang.Object` parameter. Default: `set` Required? No

6.3.1.1.2 Add get and set Methods to the Bean Add `get(String)` and `set(String, Object)` methods to the bean. The `get()` method returns a value by property name and the `set()` method stores a value by property name. The default names for these methods are `get` and `set`, and they can be overridden with the `@XmlVirtualAccessMethods` annotation.

EclipseLink weaves these methods if weaving is enabled, which provides support for lazy loading, change tracking, fetch groups, and internal optimizations.

6.3.1.1.3 Add a Data Structure Add a data structure to store the extended attributes and values, that is, the virtual mappings. These can then be mapped to the database. See "Task 2: Provide Additional Mappings".

A common way to store the virtual mappings is in a `Map`, but you can use other ways, as well. For example you could store the virtual mappings in a directory system.

When using field-based access, annotate the data structure with `@XmlTransient` so it cannot use it for another mapping. When using property-based access, `@XmlTransient` is unnecessary.

6.3.1.1.4 Use XML As an alternative to, or in addition to, using `@XmlVirtualAccessMethods`, you can use the `<access>` and `<access-methods>` elements, for example:

```
<access>VIRTUAL</access>
<access-methods set-method="get" get-method="set"/>
```

XML to enable virtual access methods using `get` and `set`:

```
<xml-virtual-access-methods/>
```

XML to enable virtual access methods using `put` instead of `set` (default):

```
<xml-virtual-access-methods set-method="put"/>
```

XML to enable virtual access methods using `retrieve` instead of `get` (default):

```
<xml-virtual-access-methods get-method="retrieve"/>
```

XML to enable virtual access methods using `retrieve` and `put` instead of `get` and `set` (default):

```
<xml-virtual-access-methods get-method="retrieve" set-method="put"/>
```

6.3.1.2 Task 2: Provide Additional Mappings

To provide additional mappings, add the mappings to the `eclipselink-oxm.xml` file, for example:

```
<xml-element java-attribute="idNumber"/>
```

6.3.2 Code Examples

The examples in this section illustrate how to use extensible JAXB beans. The example begins with the creation of a base class that other classes can extend. In this case the extensible classes are for `Customers` and `PhoneNumbers`. Mapping files are created for two separate tenants. Even though both tenants share several real properties, they will define virtual properties that are unique to their requirements.

6.3.2.1 Basic Setup

Example 6–5 illustrates a base class, `ExtensibleBase`, which other extensible classes can extend. In the example, the use of the `@XmlTransient` annotation prevents `ExtensibleBase` from being mapped as an inheritance relationship. The real properties represent the parts of the model that will be common to all tenants. The per-tenant extensions will be represented as virtual properties.

Example 6–5 A Base Class for Extensible Classes

```
package examples.virtual;

import java.util.HashMap;
import java.util.Map;

import javax.xml.bind.annotation.XmlTransient;

import org.eclipse.persistence.oxm.annotations.XmlVirtualAccessMethods;

@XmlTransient
@XmlVirtualAccessMethods(setMethod="put")
public class ExtensibleBase {

    private Map<String, Object> extensions = new HashMap<String, Object>();

    public <T> T get(String property) {
        return (T) extensions.get(property);
    }

    public void put(String property, Object value) {
        extensions.put(property, value);
    }
}
```

Example 6–6 illustrates the definition of a `Customer` class. The `Customer` class is extensible because it inherits from a domain class that has been annotated with `@XmlVirtualAccessMethods`.

Example 6–6 An Extensible Customer Class

```
package examples.virtual;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
```

```
public class Customer extends ExtensibleBase {

    private String firstName;
    private String lastName;
    private Address billingAddress;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Address getBillingAddress() {
        return billingAddress;
    }

    public void setBillingAddress(Address billingAddress) {
        this.billingAddress = billingAddress;
    }

}
```

Example 6–7 illustrates an `Address` class. It is not necessary for every class in your model to be extensible. In this example, the `Address` class does not have any virtual properties.

Example 6–7 A Nonextensible Address Class

```
package examples.virtual;

public class Address {

    private String street;

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

}
```

Example 6–8 illustrates a `PhoneNumber` class. Like `Customer`, `PhoneNumber` will be an extensible class.

Example 6–8 An Extensible PhoneNumber Class

```
package examples.virtual;
```

```
import javax.xml.bind.annotation.XmlValue;

public class PhoneNumber extends ExtensibleBase {

    private String number;

    @XmlValue
    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

}
```

6.3.2.2 Define the Tenants

The examples in this section define two separate tenants. Even though both tenants share several real properties, the corresponding XML representation can be quite different due to virtual properties.

Tenant 1

The first tenant is an online sporting goods store that requires the following extensions to its model:

- Customer ID
- Customer's middle name
- Shipping address
- A collection of contact phone numbers
- Type of phone number (that is, home, work, or cell)

The metadata for the virtual properties is supplied through Object-XML's XML mapping file. Virtual properties are mapped in the same way as real properties. Some additional information is required, including type (since this cannot be determined through reflection), and for collection properties, a container type. The virtual properties defined below for `Customer` are `middleName`, `shippingAddress`, and `phoneNumbers`. For `PhoneNumber`, the virtual property is the `type` property.

Example 6–9 illustrates the `binding-tenant1.xml` mapping file.

Example 6–9 Defining Virtual Properties for Tenant 1

```
<?xml version="1.0"?>
<xml-bindings
    xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
    package-name="examples.virtual">
    <java-types>
        <java-type name="Customer">
            <xml-type prop-order="firstName middleName lastName billingAddress
shippingAddress phoneNumbers"/>
            <java-attributes>
                <xml-attribute
                    java-attribute="id"
                    type="java.lang.Integer"/>
                <xml-element
```

```
                    java-attribute="middleName"
                    type="java.lang.String"/>
                <xml-element
                    java-attribute="shippingAddress"
                    type="examples.virtual.Address"/>
                <xml-element
                    java-attribute="phoneNumbers"
                    name="phoneNumber"
                    type="examples.virtual.PhoneNumber"
                    container-type="java.util.List"/>
            </java-attributes>
        </java-type>
        <java-type name="PhoneNumber">
            <java-attributes>
                <xml-attribute
                    java-attribute="type"
                    type="java.lang.String"/>
            </java-attributes>
        </java-type>
    </java-types>
</xml-bindings>
```

The `get` and `set` methods are used on the domain model to interact with the real properties and the accessors defined on the `@XmlVirtualAccessMethods` annotation are used to interact with the virtual properties. The normal JAXB mechanisms are used for marshal and unmarshal operations. Example 6–10 illustrates the `Customer` class code for tenant 1 to obtain the data associated with virtual properties.

Example 6–10 Tenant 1 Code to Provide the Data Associated with Virtual Properties

```
...
Customer customer = new Customer();

//Set Customer's real properties
customer.setFirstName("Jane");
customer.setLastName("Doe");

Address billingAddress = new Address();
billingAddress.setStreet("1 Billing Street");
customer.setBillingAddress(billingAddress);

//Set Customer's virtual 'middleName' property
customer.put("middleName", "Anne");

//Set Customer's virtual 'shippingAddress' property
Address shippingAddress = new Address();
shippingAddress.setStreet("2 Shipping Road");
customer.put("shippingAddress", shippingAddress);

List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();
customer.put("phoneNumbers", phoneNumbers);

PhoneNumber workPhoneNumber = new PhoneNumber();
workPhoneNumber.setNumber("555-WORK");
//Set the PhoneNumber's virtual 'type' property
workPhoneNumber.put("type", "WORK");
phoneNumbers.add(workPhoneNumber);

PhoneNumber homePhoneNumber = new PhoneNumber();
```

```
homePhoneNumber.setNumber("555-HOME");
//Set the PhoneNumber's virtual 'type' property
homePhoneNumber.put("type", "HOME");
phoneNumbers.add(homePhoneNumber);

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextFactory.ECLIPSELINK_OXM_XML_KEY,
"examples/virtual/binding-tenant1.xml");
JAXBContext jc = JAXBContext.newInstance(new Class[] {Customer.class,
Address.class}, properties);

Marshaller marshaller = jc.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(customer, System.out);
...
```

Example 6–11 illustrates the XML output from the `Customer` class for tenant 1.

Example 6–11 XML Output from the Customer Class for Tenant 1

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
   <firstName>Jane</firstName>
   <middleName>Anne</middleName>
   <lastName>Doe</lastName>
   <billingAddress>
      <street>1 Billing Street</street>
   </billingAddress>
   <shippingAddress>
      <street>2 Shipping Road</street>
   </shippingAddress>
   <phoneNumber type="WORK">555-WORK</phoneNumber>
   <phoneNumber type="HOME">555-HOME</phoneNumber>
</customer>
```

Tenant 2

The second tenant is a streaming media provider that offers on-demand movies and music to its subscribers. It requires a different set of extensions to the core model:

- A single contact phone number

For this tenant, the mapping file is also used to customize the mapping of the real properties.

Example 6–12 illustrates the `binding-tenant2.xml` mapping file.

Example 6–12 Defining Virtual Properties for Tenant 2

```
<?xml version="1.0"?>
<xml-bindings
    xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
    package-name="examples.virtual">
    <xml-schema namespace="urn:tenant1" element-form-default="QUALIFIED"/>
    <java-types>
        <java-type name="Customer">
            <xml-type prop-order="firstName lastName billingAddress phoneNumber"/>
            <java-attributes>
                <xml-attribute java-attribute="firstName"/>
                <xml-attribute java-attribute="lastName"/>
                <xml-element java-attribute="billingAddress" name="address"/>
                <xml-element
```

```
                    java-attribute="phoneNumber"
                    type="examples.virtual.PhoneNumber"/>
            </java-attributes>
        </java-type>
    </java-types>
</xml-bindings>
```

Example 6–13 illustrates the tenant 2 `Customer` class code to obtain the data associated with virtual properties.

Example 6–13 Tenant 2 Code to Provide the Data Associated with Virtual Properties

```
...
Customer customer = new Customer();
customer.setFirstName("Jane");
customer.setLastName("Doe");

Address billingAddress = new Address();
billingAddress.setStreet("1 Billing Street");
customer.setBillingAddress(billingAddress);

PhoneNumber phoneNumber = new PhoneNumber();
phoneNumber.setNumber("555-WORK");
customer.put("phoneNumber", phoneNumber);

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextFactory.ECLIPSELINK_OXM_XML_KEY,
"examples/virtual/binding-tenant2.xml");
JAXBContext jc = JAXBContext.newInstance(new Class[] {Customer.class,
Address.class}, properties);

Marshaller marshaller = jc.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(customer, System.out);
...
```

Example 6–14 illustrates the XML output from the `Customer` class for tenant 2.

Example 6–14 XML Output from the Customer Class for Tenant 2

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="urn:tenant1" firstName="Jane" lastName="Doe">
   <address>
      <street>1 Billing Street</street>
   </address>
   <phoneNumber>555-WORK</phoneNumber>
</customer>
```

6.4 Using Single-Table Multi-Tenancy

A key element to implementing SaaS is the ability for multiple application tenants to use a shared persistence schema while ensuring that the tenant only works on its own data. Single-table multi-tenancy uses a single-table for all application tenants and differentiates application tenants based on tenant discriminator columns with specific application context values. Applications can configure as many discriminator columns as needed or rely on default behavior. For additional details on single-table multi-tenancy, see:

http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Single-Table_Multi-Tenancy

The following topics are included in this section:

- Section 6.4.1, "Main Tasks"
- Section 6.4.2, "Additional Resources"

6.4.1 Main Tasks

The tasks in this section provide instructions for using single-table multi-tenancy when creating applications that are designed to run in SaaS environments.

The following tasks are included in this section:

- Task 1: Enable Single-Table Multi-Tenancy
- Task 2: Specify Tenant Discriminator Columns
- Task 3: Use the Discriminator Column at Run Time

6.4.1.1 Task 1: Enable Single-Table Multi-Tenancy

Single-table multi-tenancy can be enabled declaratively using the `@Multitenant` annotation; or in an ORM XML file using the `<multitenant>` element; or using annotations and XML together.

Using the @Multitenant Annotation

To use the `@Multitenant` annotation, include the annotation with an `@Entity` or `@MappedSuperclass` annotation and include the `SINGLE_TABLE` attribute. For example:

```
@Entity
@Multitenant(SINGLE_TABLE)
public class Employee {
}
```

The `SINGLE_TABLE` attributes states that the table or tables (`Table` and `SecondaryTable`) associated with the given entity can be shared among tenants.

Using the <multitenant> Element

To use the `<multitenant>` element, include the element within an `<entity>` element. For example:

```
<entity class="model.Employee">
   <multitenant type="SINGLE_TABLE">
   ...
   </multitenant>
   ...
</entity>
```

6.4.1.2 Task 2: Specify Tenant Discriminator Columns

Discriminator columns are used together with an associated application context to indicate which rows in a table an application tenant can access. Multiple tenant discriminator columns can be specified. If no tenant discriminator column is specified a default column named `TENANT_ID` is used along with the default `eclipselink.tenant-id` context property. The tenant discriminator column is

assumed to be on the primary table unless a table or secondary table is explicitly specified. To change the default behavior, see:

http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Single-Table_Multi-Tenancy#Defining_Persistence_Unit_and_Entity_Mappings_Defaults

Tenant discriminator columns can be specified declaratively using the `@TenantDiscriminatorColumn` or `@TenantDiscriminatorColumns` annotations; or in an ORM XML file using the `<tenant-discriminator-column>` element.

Using the @TenantDiscriminatorColumn Annotation

To use the `@TenantDiscriminatorColumn` annotation, include the annotation with an `@Entity` or `@MappedSuperclass` annotation and include the `name` and `contextProperty` attributes. For example:

```
@Entity
@Multitenant(SINGLE_TABLE)
@TenantDiscriminatorColumn(name = "TENANT", contextProperty = "multi-tenant.id")
public class Employee {
}
```

To specify multiple columns, include multiple `@TenantDiscriminatorColumn` annotations within the `@TenantDiscriminatorColumns` annotation and include the table where the column is located if it is not located on the primary table. For example:

```
@Entity
@Table(name = "EMPLOYEE")
@SecondaryTable(name = "RESPONSIBILITIES")
@Multitenant(SINGLE_TABLE)
@TenantDiscriminatorColumns({
   @TenantDiscriminatorColumn(name = "TENANT_ID",
      contextProperty = "employee-tenant.id", length = 20)
   @TenantDiscriminatorColumn(name = "TENANT_CODE",
      contextProperty = "employee-tenant.code", discriminatorType = STRING,
      table = "RESPONSIBILITIES")
  }
)
public Employee() {
   ...
}
```

Using the <tenant-discriminator-column> Element

To use the `<tenant-discriminator-column>` element, include the element within a `<multitenant>` element and include the `name` and `context-property` attributes. For example:

```
<entity class="model.Employee">
   <multitenant>
      <tenant-discriminator-column name="TENANT"
         context-property="multi-tenant.id"/>
   </multitenant>
   ...
</entity>
```

To specify multiple columns, include additional <tenant-discriminator-column> elements and include the table where the column is located if it is not located on the primary table. For example:

```
<entity class="model.Employee">
   <multitenant type="SINGLE_TABLE">
      <tenant-discriminator-column name="TENANT_ID"
         context-property="employee-tenant.id" length="20"/>
      <tenant-discriminator-column name="TENANT_CODE"
         context-property="employee-tenant.id" discriminator-type="STRING"
         table="RESPONSIBILITIES"/>
   </multitenant>
   <table name="EMPLOYEE"/>
   <secondary-table name="RESPONSIBILITIES"/>
   ...
</entity>
```

Mapping Tenant Discriminator Columns

Tenant discriminator columns can be mapped to a primary key or another column. The following example maps the tenant discriminator column to the primary key on the table during DDL generation:

```
@Entity
@Table(name = "ADDRESS")
@Multitenant
@TenantDiscriminatorColumn(name = "TENANT", contextProperty = "tenant.id",
   primaryKey = true)
public Address() {
  ...
}
```

To have the discriminator column mapped to the primary key as part of the object entity, the column must be mapped. For example

```
@Id
@Column("TENANT")
public int tenant;
```

The following example maps the tenant discriminator column to a primary key in the ORM XML file:

```
<entity class="model.Address">
   <multitenant>
      <tenant-discriminator-column name="TENANT"
         context-property="multi-tenant.id" primary-key="true"/>
   </multitenant>
   <table name="ADDRESS"/>
   ...
</entity>
```

The following example maps the tenant discriminator column to another column name AGE.

```
@Entity
@Table(name = "Player")
@Multitenant
@TenantDiscriminatorColumn(name = "AGE", contextProperty = "tenant.age")
public Player() {
  ...
```

```
  @Basic
  @Column(name="AGE", insertable="false", updatable="false")
  public int age;
}
```

Or, in the ORM XML file as follows:

```
<entity class="model.Player">
  <multi-tenant>
    <tenant-discriminator-column name="AGE" context-property="tenant.age"/>
  </multi-tenant>
  <table name="PLAYER"/>
  ...
  <attributes>
    <basic name="age" insertable="false" updatable="false">
      <column name="AGE"/>
    </basic>
    ...
  </attributes>
  ...
</entity>
```

Specifying a context property at Run Time

At runtime, the context property configuration can be specified using a persistence unit definition that is passed to a create entity manager factory call or set on an individual entity manager. For example

```
<persistence-unit name="multi-tenant">
   ...
   <properties>
      <property name="tenant.id" value="707"/>
      ...
   </properties>
</persistence-unit>
```

Or, alternatively in code as follows:

```
HashMap properties = new HashMap();
properties.put(PersistenceUnitProperties.MULTITENANT_PROPERTY_DEFAULT, "707");
EntityManager em = Persistence.createEntityManagerFactory("multi-tenant-pu",
   properties).createEntityManager();
```

An entity manager property definition follows:

```
EntityManager em =
  Persistence.createEntityManagerFactory("multi-tenant-pu").createEntityManager();
em.beginTransaction();
em.setProperty("other.tenant.id.property", "707");
em.setProperty(EntityManagerProperties.MULTITENANT_PROPERTY_DEFAULT, "707");
...
```

6.4.1.3 Task 3: Use the Discriminator Column at Run Time

The tenant discriminator column can be used at run time through entity manager operations and querying. The tenant discriminator column and value are supported through the following entity manager operations:

- `persist()`
- `find()`

- `refresh()`

The tenant discriminator column and value are supported through the following queries:

- Named queries
- Update all
- Delete all

> **Note:** Multi-tenancy is not supported through named native queries. To use named native queries in a multi-tenant environment, manually handle any multi-tenancy issues directly in the query. In general, it is best to avoid named native queries in a multi-tenant environment.

6.4.2 Additional Resources

The following additional resources are available:

- Section 6.4.2.1, "Code Samples"
- Section 6.4.2.2, "Related Javadoc"

6.4.2.1 Code Samples

`http://wiki.eclipse.org/EclipseLink/Examples/JPA/Multitenant`

`http://wiki.eclipse.org/EclipseLink/Examples/MySports`

`http://wiki.eclipse.org/EclipseLink/Examples/JPA/Multitenant/VPD`

6.4.2.2 Related Javadoc

For more information, see the following APIs in *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

- `org.eclipse.persistence.annotations.Multitenant`
- `org.eclipse.persistence.annotations.TenantDiscriminatorColumn`
- `org.eclipse.persistence.annotations.TenantDiscriminatorColumns`

6.5 Using an External Metadata Source

With TopLink, you can store your mapping information in a metadata source that is external to the running application. Because the mapping information is retrieved when the application creates the persistence unit, you can dynamically override or extend mappings in a deployed application.

6.5.1 Using the eclipselink-orm.xml File Externally

With TopLink, you can use the `eclipselink-orm.xml` file to support advanced mapping types and options. This file can override the standard JPA `orm.xml` mapping configuration file.

6.5.2 Main Tasks

This section includes the following tasks:

- Task 1: Configure the Persistence Unit

- Task 2: Configure the Server

6.5.2.1 Task 1: Configure the Persistence Unit

You can configure your persistence unit to use the external metadata by:

- Section 6.5.2.1.1, "Accessing a Fixed Location"
- Section 6.5.2.1.2, "Accessing an Application Context Based Location"

6.5.2.1.1 Accessing a Fixed Location The easiest way to access an external file, such as the `eclipselink-orm.xml` file with additional mapping information, is by making the file available from a fixed URL on the web server.

Use the `eclipselink.metadata-source.xml.url` property, as shown in Example 6–15, to specify the location:

Example 6–15 Fixed Location

```
<property name="eclipselink.metadata-source" value="XML"/>
<property name="eclipselink.metadata-source.xml.url"
value="http://myserverlocation/"/>
```

6.5.2.1.2 Accessing an Application Context Based Location For more complex requirements, such as providing tenant-specific extensions in a multi-tenant application, you can specify the location of the external metadata based on the application context.

Implement the `MetadataSource` interface, as shown in Example 6–16, to specify the location:

Example 6–16 Fixed Location

```
<property name="eclipselink.metadata-source" value="mypackage.MyMetadataSource"/>
<property name="eclipselink.metadata-source.xml.url" value="foo://bar"/>
```

Example 6–17 illustrates how to return a specific mapping file, based on tenant:

Example 6–17 Tenant-specific Mapping File

```
public class AdminMetadataSource extends XMLMetadataSource {

    @Override
    public XMLEntityMappings getEntityMappings(Map<String, Object> properties,
ClassLoader classLoader, SessionLog log) {
        String leagueId = (String) properties.get(LEAGUE_CONTEXT);
        properties.put(PersistenceUnitProperties.METADATA_SOURCE_XML_URL,
"http://myserverlocation/rest/" + leagueId + "/orm");
        return super.getEntityMappings(properties, classLoader, log);
    }
}
```

6.5.2.2 Task 2: Configure the Server

To access the metadata file, the server must provide URL access to the mapping file by using:

- Static file serving,

- A server-based solution with its own mapping file or a mapping file built on-demand from stored mapping information,
- Or some other web technology.

6.5.3 Additional Resources

For additional information on JPA deployment, see the following sections of the JPA Specification (`http://jcp.org/en/jsr/detail?id=317`):

- Section 7.2, "Bootstrapping in Java SE Environments"
- Chapter 7, "Container and Provider Contracts for Deployment and Bootstrapping"

6.5.3.1 Javadoc

For more information, see the following APIs in *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

- `PersistenceUnitProperties` class

7 Mapping JPA to XML

You can use the Java Architecture for XML Binding (JAXB) and its Mapping Objects to XML (MOXy) extensions to map JPA entities to XML. Mapping JPA entities to XML is useful when you want to create a data access service with Java API for RESTful Web Services (JAX-RS), Java API for XML Web Services (JAX-WS), or Spring.

This chapter demonstrates some typical techniques for mapping JPA entities to XML. It contains the following sections:

7.1 Understanding JPA-to-XML Mapping Concepts

Working with the examples that follow requires some understanding of such high-level JPA-to-XML mapping concepts, such as JAXB, MOXy, XML binding, and how to override JAXB annotations. The following sections will give you a basic understanding of these concepts:

7.1.1 XML Binding

XML binding is how you represent information in an XML document as an object in computer memory. This allows applications to access the data in the XML from the object rather than using the Domain Object Model (DOM), the Simple API for XML (SAX) or the Streaming API for XML (StAX) to retrieve the data from a direct representation of the XML itself. When binding, JAXB applies a tree structure to the graph of JPA entities. Multiple tree representations of a graph are possible and will depend on the root object chosen and the direction the representations are traversed.

You can find examples of XML binding with JAXB in Section 7.2, "Binding JPA Entities to XML".

7.1.2 JAXB

JAXB is a Java API that allows a Java program to access an XML document by presenting that document to the program in a Java format. This process, called binding, represents information in an XML document as an object in computer memory. In this way, applications can access the data in the XML from the object rather than using the Domain Object Model (DOM) or the Streaming API for XML (SAX) to retrieve the data from a direct representation of the XML itself. Usually, an XML binding is used with JPA entities to create a data access service by optimizing a JAX-WS or JAX-RS implementation. Both of these web service standards use JAXB as the default binding layer. This service provides a means to access data exposed by JPA across computers, where the client computer might or might not be using Java.

JAXB uses an extended set of annotations to define the binding rules for Java-to-XML mapping. These annotations are subclasses of the `javax.xml.bind.*` packages in the Oracle TopLink API. For more information on these annotations, see *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

For more information about JAXB, see "Java Architecture for XML Binding (JAXB)" at:

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

7.1.3 MOXy

MOXy implements JAXB for TopLink. It allows you to map a Plain Old Java Objects (POJO) model to an XML schema, greatly enhancing your ability to create JPA-to-XML mappings. MOXy supports all the standard JAXB annotations in the `javax.xml.bind.annotation` package plus has its own extensions in the `org.eclipse.persistence.oxm.annotations` package. You can use these latter annotations in conjunction with the standard annotations to extend the utility of JAXB. Because MOXy represents the optimal JAXB implementation, you still implement it whether or not you explicitly use any of its extensions. MOXy offers these benefits:

- It allows you to map your own classes to your own XML schema, a process called "Meet in the Middle Mapping". This avoids static coupling of your mapped classes with a single XML schema.
- It offers specific features, such as compound key mapping and mapping relationships with back-pointers to address critical JPA-to-XML mapping issues.
- It allows you to map your existing JPA models to industry standard schemas.
- It allows you to combine MOXy mappings and the TopLink persistence framework to interact with your data through the J2EE connector architecture (JCA).
- It offers superior performance in several mapping scenarios.

For more information on MOXy, see the MOXy FAQ at:

`http://wiki.eclipse.org/EclipseLink/FAQ/WhatIsMOXy`

7.1.4 XML Data Representation

JAXB/MOXy is not always the most effective way to map JPA to XML. For example, you would not use JAXB if:

- You want to specify metadata for a third-party class but do not have access to the source.
- You want to map an object model to multiple XML schemas, because JAXB rules preclude applying more than one mapping by using annotations.
- Your object model already contains too many annotations—for example, from such services as JPA, Spring, JSR-303, and so on—and you want to specify the metadata elsewhere.

Under these and similar circumstances, you can use an XML data representation by exposing the `eclipselink_oxm.xml` file.

XML metadata works in two modes:

- It adds to the metadata supplied by annotations. This is useful when:
  - Annotations define version one of the XML representation, and you use XML metadata to change the metadata for future versions.
  - You use the standard JAXB annotations, and use the XML metadata for the MOXy extensions. In this way you do not introduce new compile time dependencies in the object model.
- It completely replaces the annotation metadata, which is useful when you want to map to different XML representations.

For more information about how to use XML data representation, see Section 7.4, "Main Tasks for Using XML Metadata Representation to Override JAXB Annotations".

7.2 Binding JPA Entities to XML

The following tasks demonstrate how to bind JPA entities to XML by using JAXB annotations. For more information about binding, see Section 7.1.1, "XML Binding"; for more information about JAXB, see Section 7.1.2, "JAXB"

- Section 7.2.1, "Main Tasks for Binding JPA Relationships to XML"
- Section 7.2.2, "Main Tasks for Binding Compound Primary Keys to XML"
- Section 7.2.3, "Main Tasks for Binding Embedded ID Classes to XML"

7.2.1 Main Tasks for Binding JPA Relationships to XML

The following tasks demonstrate how to use JAXB to derive an XML representation from a set of JPA entities, a process called binding (read about XML binding in Section 7.2, "Binding JPA Entities to XML"). These tasks will show how to bind two common JPA relationships to map an `Employee` entity to that employee's telephone number, address, and department:

- Privately-owned relationships
- Shared reference relationships

The tasks for binding JPA relationships to XML are the following:

- Task 1: Define the Accessor Type and Import Packages
- Task 2: Map Privately Owned Relationships
- Task 3: Map the Shared Reference Relationship

7.2.1.1 Task 1: Define the Accessor Type and Import Packages

Because all of the following examples use the same accessor type, `FIELD`, define it at the package level by using the JAXB annotation `@XmlAccessorType`. At this point, you would also import the necessary packages:

```
@XmlAccessorType(XmlAccessType.FIELD)
package com.example.model;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
```

7.2.1.2 Task 2: Map Privately Owned Relationships

A privately owned relationship occurs when the target object is referenced only by a single source object. This type of relationship can be either one-to-one and embedded, or one-to-many.

This task shows how to create bidirectional mappings for both of these types of relationships between the `Employee` entity and the `Address` and `PhoneNumber` entities.

7.2.1.2.1 Mapping a One-to-One and Embedded Relationship

The JPA `@OneToOne` and `@Embedded` annotations indicate that only one instance of the source entity is able to refer to the same target entity instance. This example shows how to map the `Employee` entity to the `Address` entity and then back to the `Employee` entity. This is considered a one-to-one mapping because the employee can be associated with only one address. Because this relationship is bidirectional, that is, `Employee` points to `Address`, which must point back to `Employee`, it uses the TopLink extension `@XmlInverseReference` to represent the back-pointer.

To create the one-to-one and embedded mapping:

1. Ensure that the accessor type `FIELD` has been defined at the package level, as described in Section 7.2.1.1, "Task 1: Define the Accessor Type and Import Packages".
2. Map one direction of the relationship, in this case the `employee` property on the `Address` entity, by inserting the `@OneToOne` annotation in the `Employee` entity:

   ```
   @OneToOne(mappedBy="resident")
   private Address residence;
   ```

   The `mappedBy` argument indicates that the relationship is owned by the `resident` field.
3. Map the return direction, that is, the `address` property on the `Employee` entity by inserting the `@OneToOne` and `@XmlInverseMapping` annotations into the `Address` entity:

   ```
   @OneToOne
   @JoinColumn(name="E_ID")
   @XmlInverseReference(mappedBy="residence")
   private Employee resident;
   ```

   The `mappedBy` field indicates that this relationship is owned by the `residence` field. `@JoinColumn` identifies the column that will contain the foreign key.

The entities should look like those shown in Example 7–1 and Example 7–2.

7.2.1.2.2 Mapping a One-to-Many Relationship The JPA `@OneToMany` annotation indicates that a single instance of the source entity can refer to multiple instances of the same target entity. For example, one employee can have multiple telephone numbers, such as a land line, a mobile number, a desired contact number, and an alternative workplace number. Each different number would be an instance of the `PhoneNumber` entity and a single `Employee` entity could point to each instance.

This task maps the employee to one of that employee's telephone numbers and back. Because the relationship between `Employee` and `PhoneNumber` is bidirectional, the example again uses the TopLink extension `@XmlInverseReference` to map the back-pointer.

To create a one-to-many mapping:

1. Ensure that the accessor type `FIELD` has been defined at the package level, as described in Section 7.2.1.1, "Task 1: Define the Accessor Type and Import Packages".
2. Map one direction of the relationship, in this case the `employee` property on the `PhoneNumber` entity, by inserting the `@OneToMany` annotation in the `Employee` entity:

   ```
   @OneToMany(mappedBy="contact")
   private List<PhoneNumber> contactNumber;
   ```

   The `mappedBy` field indicates that this relationship is owned by the `contact` field.
3. Map the return direction, that is the telephone number property on the `Employee` entity, by inserting the `@ManyToOne` and `@XmlInverseMapping` annotations into the `PhoneNumber` entity:

   ```
   @ManyToOne
   @JoinColumn(name="E_ID", referencedColumnName = "E_ID")
   @XmlInverseReference(mappedBy="contactNumber")
   private Employee contact;
   ```

   The `mappedBy` field indicates that this relationship is owned by the `contactNumber` field. The `@JoinColumn` annotation identifies the column that will contain the foreign key (`name="E_ID"`) and the column referenced by the foreign key (`referencedColumnName = "E_ID"`).

The entities should look like those shown in Example 7–1 and Example 7–3.

7.2.1.3 Task 3: Map the Shared Reference Relationship

A shared reference relationship occurs when target objects are referenced by multiple source objects. For example, a business might be segregated into multiple departments, such as IT, human resources, finance, and so on. Each of these departments has multiple employees of differing job descriptions, pay grades, locations, and so on. Managing departments and employees requires shared reference relationships.

Because a shared reference relationship cannot be safely represented as nesting in XML, use key relationships. To specify the ID fields on JPA entities, use the TopLink JAXB `@XmlID` annotation on non-string fields and properties and `@XmlIDREF` on string fields and properties.

The following examples how to map a many-to-one shared reference relationship and a many-to-many shared reference relationship.

7.2.1.3.1 Mapping a Many-to-One Shared Reference Relationship In a many-to-one mapping, one or more instances of the source entity are able to refer to the same target entity instance. This example demonstrates how to map an employee to one of that employee's multiple telephone numbers.

To map a many-to-one shared reference relationship:

1. Ensure that the accessor type `FIELD` has been defined at the package level, as described in Section 7.2.1.1, "Task 1: Define the Accessor Type and Import Packages".
2. Map one direction of the relationship, in this case the phone number property on the `Employee` entity, by inserting the `@ManyToOne` annotation in the `PhoneNumber` entity:

   ```
   @ManyToOne
   @JoinColumn(name="E_ID", referencedColumnName = "E_ID")
   @XmlIDREF
   private Employee contact;
   ```

   The `@JoinColumn` annotation identifies the column that will contain the foreign key (`name="E_ID"`) and the column referenced by the foreign key (`referencedColumnName = "E_ID"`). The `@XmlIDREF` annotation indicates that this will be the primary key for the corresponding table.
3. Map the return direction, that is the employee property on the `PhoneNumber` entity, by inserting the `@OneToMany` and `@XmlInverseMapping` annotations into the `Address` entity:

   ```
   @OneToMany(mappedBy="contact")
   @XmlInverseReference(mappedBy="contact")
   private List<PhoneNumber> contactNumber;
   ```

   The `mappedBy` field for both annotations indicates that this relationship is owned by the `contact` field.

The entities should look like those shown in Example 7–1 and Example 7–3.

7.2.1.3.2 Mapping a Many-to-Many Shared Reference Relationship The `@ManyToMany` annotation indicates that one or more instances of the source entity are able to refer to one or more target entity instances. Because the relationship between `Department` and `Employee` is bidirectional, this example again uses the TopLink `@XmlInverseReference` annotation to represent the back-pointer.

To map a many-to-many shared reference relationship, do the following:

1. Ensure that the accessor type `FIELD` has been defined at the package level, as described in Section 7.2.1.1, "Task 1: Define the Accessor Type and Import Packages".
2. Create a `Department` entity by inserting the following code:

   ```
   @Entity
   public class Department {
   ```

3. Under this entity, define the many-to-many relationship and the entity's join table by inserting the following code:

   ```
   @ManyToMany
   @JoinTable(name="DEPT_EMP", joinColumns =
       @JoinColumn(name="D_ID", referencedColumnName = "D_ID"),
           inverseJoinColumns = @JoinColumn(name="E_ID",
               referencedColumnName = "E_ID"))
   ```

This code creates a join table called `DEPT_EMP` and identifies the column that will contain the foreign key (`name="E_ID"`) and the column referenced by the foreign key (`referencedColumnName = "E_ID"`). Additionally, it identifies the primary table on the inverse side of the relationship.

4. Complete the initial mapping, in this case the `Department` entity's `employee` property, and make it a foreign key for this entity by inserting the following code:

```
@XmlIDREF
private List<Employee> member;
```

5. In the `Employee` entity created in Section 7.2.1.2.1, "Mapping a One-to-One and Embedded Relationship", specify that `eId` is the primary key for JPA (the `@Id` annotation), and for JAXB (the `@XmlID` annotation) by inserting the following code:

```
@Id
@Column(name="E_ID")
@XmlID
private BigDecimal eId;
```

6. Still within the `Employee` entity, complete the return mapping by inserting the following code:

```
@ManyToMany(mappedBy="member")
@XmlInverseReference(mappedBy="member")
private List<Department> team;
```

The entities should look like those shown in Example 7–1 and Example 7–4.

7.2.1.4 JPA Entities

After the mappings are created, the entities should look like those in the following examples:

- Example 7–1, "Employee Entity"
- Example 7–2, "Address Entity"
- Example 7–3, "PhoneNumber Entity"
- Example 7–4, "Department Entity"

> **Note:** To save space, package names, import statements, and the `get` and `set` methods have been omitted from the code examples. All examples use standard JPA annotations.

Example 7–1 Employee Entity

```
@Entity
public class Employee {

    @Id
    @Column(name="E_ID")
    private BigDecimal eId;

    private String name;

    @OneToOne(mappedBy="resident")
    private Address residence;
```

```
    @OneToMany(mappedBy="contact")
    private List<PhoneNumber> contactNumber;

    @ManyToMany(mappedBy="member")
    private List<Department> team;

}
```

Example 7–2 Address Entity

```
@Entity
public class Address {

    @Id
    @Column(name="E_ID", insertable=false, updatable=false)
    private BigDecimal eId;

    private String city;

    private String street;

    @OneToOne
    @JoinColumn(name="E_ID")
    private Employee resident;

}
```

Example 7–3 PhoneNumber Entity

```
@Entity
@Table(name="PHONE_NUMBER")
public class PhoneNumber {

    @Id
    @Column(name="P_ID")
    private BigDecimal pId;

    @ManyToOne
    @JoinColumn(name="E_ID", referencedColumnName = "E_ID")
    private Employee contact;

    private String num;

}
```

Example 7–4 Department Entity

```
@Entity
public class Department {

    @Id
    @Column(name="D_ID")
    private BigDecimal dId;

    private String name;

    @ManyToMany
    @JoinTable(name="DEPT_EMP", joinColumns =
        @JoinColumn(name="D_ID", referencedColumnName = "D_ID"),
            inverseJoinColumns = @JoinColumn(name="E_ID",
```

```
                referencedColumnName = "E_ID"))
    private List<Employee> member;

}
```

7.2.2 Main Tasks for Binding Compound Primary Keys to XML

When a JPA entity has compound primary keys, you can bind the entity by using JAXB annotations and certain Oracle TopLink extensions, as shown in the following tasks:

- Task1: Define the XML Accessor Type
- Task 2: Create the Target Object
- Task 3: Create the Source Object

7.2.2.1 Task1: Define the XML Accessor Type

Define the accessor type as `FIELD`, as described in Section 7.2.1.1, "Task 1: Define the Accessor Type and Import Packages".

7.2.2.2 Task 2: Create the Target Object

To create the target object, do the following:

1. Create an `Employee` entity with a composite primary key class called `EmployeeId` to map to multiple fields or properties of the entity:

   ```
   @Entity
   @IdClass(EmployeeId.class)
   public class Employee {
   ```

2. Specify the first primary key, `eId`, of the entity and map it to a column:

   ```
   @Id
   @Column(name="E_ID")
   @XmlID
   private BigDecimal eId;
   ```

3. Specify the second primary key, `country`. In this instance, you need to use `@XmlKey` to identify the primary key because only one property—here, `eId`—can be annotated with the `@XmlID` annotation.

   ```
   @Id
   @XmlKey
   private String country;
   ```

   The `@XmlKey` annotation marks a property as a key that will be referenced by using a key-based mapping via the `@XmlJoinNode` annotation in the source object. This is similar to the `@XmlKey` annotation except it does not require the property be bound to the schema type ID. This is a typical application of the `@XmlKey` annotation.

4. Create a many-to-one mapping of the `Employee` property on `PhoneNumber` by inserting the following code:

   ```
   @OneToMany(mappedBy="contact")
   @XmlInverseReference(mappedBy="contact")
   private List<PhoneNumber> contactNumber;
   ```

The `Employee` entity should look like that shown in Example 7–5.

Example 7–5 Employee Entity with Compound Primary Keys

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {

    @Id
    @Column(name="E_ID")
    @XmlID
    private BigDecimal eId;

    @Id
    @XmlKey
    private String country;

    @OneToMany(mappedBy="contact")
    @XmlInverseReference(mappedBy="contact")
    private List<PhoneNumber> contactNumber;

}
```

7.2.2.3 Task 3: Create the Source Object

This task creates the source object, the `PhoneNumber` entity. Because the target object has a compound key, you must use the TopLink `@XmlJoinNodes` annotation to set up the mapping.

To create the source object:

1. Create the `PhoneNumber` entity:

   ```
   @Entity
   public class PhoneNumber {
   ```

2. Create a many-to-one relationship and define the join columns:

   ```
    @ManyToOne
       @JoinColumns({
           @JoinColumn(name="E_ID", referencedColumnName = "E_ID"),
           @JoinColumn(name="E_COUNTRY", referencedColumnName = "COUNTRY")
      })
   ```

3. Set up the mapping by using the TopLink `@XmlJoinNodes` annotation:

   ```
   @XmlJoinNodes( {
           @XmlJoinNode(xmlPath="contact/id/text()",
   referencedXmlPath="id/text()"),
           @XmlJoinNode(xmlPath="contact/country/text()",
   referencedXmlPath="country/text()")
       })
   ```

4. Define the `contact` property:

   ```
   private Employee contact;

   }
   ```

The target object should look like that shown in Example 7–6.

Example 7–6 PhoneNumber Entity

```
@Entity
public class PhoneNumber {
```

```
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="E_ID", referencedColumnName = "E_ID"),
        @JoinColumn(name="E_COUNTRY", referencedColumnName = "COUNTRY")
    })
    @XmlJoinNodes( {
        @XmlJoinNode(xmlPath="contact/id/text()", referencedXmlPath="id/text()"),
        @XmlJoinNode(xmlPath="contact/country/text()", referencedXmlPath="country/text()")
    })
    private Employee contact;

}
```

7.2.3 Main Tasks for Binding Embedded ID Classes to XML

An embedded ID defines a separate `Embeddable` Java class to contain the entity's primary key. It is defined through the `@EmbeddedId` annotation. The embedded ID's `Embeddable` class must define each `Id` attribute for the entity using basic mappings. In the embedded Id, all attributes in its `Embeddable` class are assumed to be part of the primary key. The following tasks show how to derive an XML representation from a set of JPA entities using JAXB when a JPA entity has an embedded ID class:

- Task 1: Define the XML Accessor Type
- Task 2: Create the Target Object
- Task 3: Implement DescriptorOrganizer as EmployeeCustomizer Class
- Task 4: Create the Source Object
- Task 5: Implement the DescriptorCustomizer as PhoneNumberCustomizer Class

7.2.3.1 Task 1: Define the XML Accessor Type

Define the XML accessor type as `FIELD`, as described in Section 7.2.1.1, "Task 1: Define the Accessor Type and Import Packages".

7.2.3.2 Task 2: Create the Target Object

The target object is an entity called `Employee` and contains the mapping for an employee's contact telephone number. Creating this target object requires implementing a `DescriptorCustomizer` interface, so you must include the TopLink `@XmlCustomizer` annotation. Also, because the relationship is bidirectional, you must implement the `@XmlInverseReference` annotation.

To create the target object:

1. Create the `Employee` entity. Use the `@IdClass` annotation to specify that the `EmployeeID` class will be mapped to multiple properties of the entity and use the `@XmlCustomizer` annotation to indicate that the class `EmployeeCustomizer` will implement the `DescriptorCustomizer` interface (see Section 7.2.3.3, "Task 3: Implement DescriptorOrganizer as EmployeeCustomizer Class").

   ```
   @Entity
   @IdClass(EmployeeId.class)
   public class Employee {
   ```

2. Define the `id` property and make it embeddable.

   ```
   @EmbeddedId
   @XmlPath(".");
   ```

```
    private EmployeeId id;
```

3. Define a one-to-many mapping, in this case, the `employee` property on the `PhoneNumber` entity. Because the relationship is bidirectional, use `@XmlInverseReference` annotation to define the return mapping. Both of these relationships will be owned by the `contact` field, as indicated by the `mappedBy` argument.

```
@OneToMany(mappedBy="contact")
@XmlInverseReference(mappedBy="contact")
private List<PhoneNumber> contactNumber;
```

The completed target object should look like that shown in Example 7–7.

Example 7–7 Employee Entity as Target Object

```
@Entity
@IdClass(EmployeeId.class)
@XmlCustomizer(EmployeeCustomizer.class)
public class Employee {

    @EmbeddedId
    private EmployeeId id;

    @OneToMany(mappedBy="contact")
    @XmlInverseReference(mappedBy="contact")
    private List<PhoneNumber> contactNumber;

}
```

7.2.3.3 Task 3: Implement DescriptorOrganizer as EmployeeCustomizer Class

In Task 2: Create the Target Object, `DescriptorCustomizer` was implemented as the class `EmployeeCustomizer`. This allows changing the XML Path (XPath) on the mapping for the `id` property to either self or "." and then specifying the XPath to the XML nodes that represent the ID. To do this:

1. Implement the `DescriptorOrganizer` class as `EmployeeOrganizer`.

```
import org.eclipse.persistence.oxm.mappings.XMLCompositeObjectMapping;

public class EmployeeCustomizer implements DescriptorCustomizer {
```

2. Specify the XPath to the XML nodes that represent the ID:

```
descriptor.addPrimaryKeyFieldName("eId/text()");
descriptor.addPrimaryKeyFieldName("country/text()");
```

The `EmployeeCustomizer` class should look like Example 7–8.

Example 7–8 EmployeeCustomizer Class with Updated XPath Information

```
import org.eclipse.persistence.config.DescriptorCustomizer;
import org.eclipse.persistence.descriptors.ClassDescriptor;
import org.eclipse.persistence.oxm.mappings.XMLCompositeObjectMapping;

public class EmployeeCustomizer implements DescriptorCustomizer {

        descriptor.addPrimaryKeyFieldName("eId/text()");
        descriptor.addPrimaryKeyFieldName("country/text()");
    }
```

```
}
```

7.2.3.4 Task 4: Create the Source Object

The source object in this task has a compound key, so you must annotate the field with the `@XmlTransient` annotation to prevent a key from being mapped by itself. Use the TopLink `@XmlCustomizer` annotation to set up the mapping.

To create the source object, do the following:

1. Create the `PhoneNumber` entity and specify another class, `PhoneNumberCustomizer`, to implement the `DescriptorCustomizer` interface.

```
@Entity
@XmlCustomizer(PhoneNumberCustomizer.class)
public class PhoneNumber {
```

2. Create a many-to-one mapping and define the join columns.

```
@ManyToOne
    @JoinColumns({
        @JoinColumn(name="E_ID", referencedColumnName = "E_ID"),
        @JoinColumn(name="E_COUNTRY", referencedColumnName = "COUNTRY")
    })
```

3. Define the `contact` property. Use the `@XmlTransient` annotation to prevent this key from being mapped by itself.

```
@XmlTransient
private Employee contact;
```

The completed `PhoneNumber` class should look like Example 7–9.

Example 7–9 PhoneNumber Class as Source Object

```
@Entity
@XmlCustomizer(PhoneNumberCustomizer.class)
public class PhoneNumber {

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="E_ID", referencedColumnName = "E_ID"),
        @JoinColumn(name="E_COUNTRY", referencedColumnName = "COUNTRY")
    })
    @XmlTransient
    private Employee contact;

}
```

7.2.3.5 Task 5: Implement the DescriptorCustomizer as PhoneNumberCustomizer Class

Code added in Task 4 indicated the need to map the `XMLObjectReferenceMapping` class to the new values. This requires implementing the `DescriptorCustomizer` class as the `PhoneNumberCustomizer` class and adding the multiple key mappings. To do this:

1. Implement `DescriptorCustomizer` as `PhoneNumberCustomizer`. Import `org.eclipse.persistence.oxm.mappings.XMLObjectReferenceMapping`:

```
import org.eclipse.persistence.oxm.mappings.XMLObjectReferenceMapping;

public class PhoneNumberCustomizer implements DescriptorCustomizer {
```

2. In the `customize` method, update the following mappings:
   - `contactMapping.setAttributeName` to `"contact"`.
   - `contactMapping.addSourceToTargetKeyFieldAssociation` to `"contact/@eID", "eId/text()"`.
   - `contactMapping.addSourceToTargetKeyFieldAssociation` to `"contact/@country", "country/text()"`.

The `PhoneNumberCustomizer` should look like that shown in Example 7–10.

Example 7–10 PhoneNumber Customizer with Updated Key Mappings

```
import org.eclipse.persistence.config.DescriptorCustomizer;
import org.eclipse.persistence.descriptors.ClassDescriptor;
import org.eclipse.persistence.oxm.mappings.XMLObjectReferenceMapping;

public class PhoneNumberCustomizer implements DescriptorCustomizer {

    public void customize(ClassDescriptor descriptor) throws Exception {
        XMLObjectReferenceMapping contactMapping = new XMLObjectReferenceMapping();
        contactMapping.setAttributeName("contact");
        contactMapping.setReferenceClass(Employee.class);
        contactMapping.addSourceToTargetKeyFieldAssociation("contact/@eID", "eId/text()");
        contactMapping.addSourceToTargetKeyFieldAssociation("contact/@country", "country/text()");
        descriptor.addMapping(contactMapping);
    }

}
```

7.2.4 Using the EclipseLink XML Binding Document

As demonstrated in the preceding tasks, TopLink implements the standard JAXB annotations to map JPA entities to an XML representation. You can also express metadata by using the EclipseLink XML Bindings document. Not only can you use XML bindings to separate your mapping information from your actual Java class, but you can also use it for more advanced metadata tasks, such as:

- Augmenting or overriding existing annotations with additional mapping information
- Specifying all mapping information externally, without using any Java annotations
- Defining your mappings across multiple EclipseLink XML Bindings documents
- Specifying *virtual* mappings that do not correspond to concrete Java fields

For more information on using the XML Bindings document, see XML Bindings in the JAXB/MOXy documentation at http://wiki.eclipse.org/EclipseLink/UserGuide/MOXy/Runtime/XML_Bindings.

7.3 Main Tasks for Mapping Simple Java Values to XML Text Nodes

There are several ways to map simple Java values directly to XML text nodes. It includes the following tasks:

- Task 1: Mapping a Value to an Attribute
- Task 2: Mapping a Value to a Text Node

7.3.1 Task 1: Mapping a Value to an Attribute

This task maps the `id` property in the Java object `Customer` to its XML representation as an attribute of the `<customer>` element. The XML will be based on the schema in Example 7–11.

Example 7–11 Example XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsd:element name="customer" type="customer-type"/>

   <xsd:complexType name="customer-type">
      <xsd:attribute name="id" type="xsd:integer"/>
   </xsd:complexType>

</xsd:schema>
```

The following procedures demonstrate how to map the `id` property from the Java object and, alternatively, how to represent the value in the Oracle TopLink Object-to-XML Mapping (OXM) metadata format.

7.3.1.1 Mapping from the Java Object

The key to creating this mapping from a Java object is the `@XmlAttribute` JAXB annotation, which maps the field to the XML attribute. To create this mapping:

1. Create the object and import `javax.xml.bind.annotation.*`:

   ```
   package example;

   import javax.xml.bind.annotation.*;
   ```

2. Declare the `Customer` class and use the `@XmlRootElement` annotation to make it the root element. Set the XML accessor type to `FIELD`:

   ```
   @XmlRootElement
   @XmlAccessorType(XmlAccessType.FIELD)
   public class Customer {
   ```

3. Map the `id` property in the `Customer` class as an attribute:

   ```
      @XmlAttribute
      private Integer id;
   ```

The object should look like that shown in Example 7–12.

Example 7–12 Customer Object with Mapped id Property

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
   @XmlAttribute
```

```
   private Integer id;

   ...
}
```

7.3.1.2 Defining the Mapping in OXM Metadata Format

If you want to represent the mapping in the TopLink OXM metadata format, use the XML tags defined in the `eclipselink-oxm.xml` file and populate them with the appropriate values, as shown in Example 7–13.

Example 7–13 Mapping id as an Attribute in OXM Metadata Format

```
...
<java-type name="Customer">
   <xml-root-element name="customer"/>
   <java-attributes>
      <xml-attribute java-attribute="id"/>
   </java-attributes>
</java-type>
...
```

For more information about the OXM metadata format, see Section 7.4, "Main Tasks for Using XML Metadata Representation to Override JAXB Annotations".

7.3.2 Task 2: Mapping a Value to a Text Node

Oracle TopLink makes it easy for you to map values from a Java object to various kinds of XML text nodes; for example, to simple text nodes, text nodes in a simple sequence, in a subset, or by position. These mappings are demonstrated in the following examples:

- Mapping a Value to a Simple Text Node
- Mapping Values to a Text Node in a Simple Sequence
- Mapping a Value to a Text Node in a Subelement
- Mapping Values to a Text Node by Position

7.3.2.1 Mapping a Value to a Simple Text Node

You can map a value from a Java object either by using JAXB annotations in the Java object or, alternatively, by representing the mapping in the TopLink OXM metadata format.

7.3.2.1.1 Mapping by Using JAXB Annotations Assume the associated schema defines an element called `<phone-number>` that accepts a string value. You can use the `@XmlValue` annotation to map a string to the `<phone-number>` node. Do the following:

1. Create the object and import `javax.xml.bind.annotation.*`:

   ```
   package example;

   import javax.xml.bind.annotation.*;
   ```

2. Declare the `PhoneNumber` class and use the `@XmlRootElement` annotation to make it the root element with the name `phone-number`. Set the XML accessor type to `FIELD`:

   ```
   @XmlRootElement(name="phone-number")
   ```

```
@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {
```

3. Insert the `@XmlValue` annotation on the line before the `number` property in the `Customer` class to map this value as an attribute:

```
@XmlValue
private String number;
```

The object should look like that shown in Example 7–14.

Example 7–14 PhoneNumber Object with Mapped number Property

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement(name="phone-number")
@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {
   @XmlValue
   private String number;

   ...
}
```

7.3.2.1.2 Defining the Mapping in OXM Metadata Format If you want to represent the mapping in the TopLink OXM metadata format, then use the XML tags defined in the `eclipselink-oxm.xml` file and populate them with the appropriate values, as shown in Example 7–15.

Example 7–15 Mapping number as an Attribute in OXM Metadata Format

```
...
<java-type name="PhoneNumber">
   <xml-root-element name="phone-number"/>
   <java-attributes>
      <xml-value java-attribute="number"/>
   </java-attributes>
</java-type>
...
```

7.3.2.2 Mapping Values to a Text Node in a Simple Sequence

You can map a sequence of values, for example a customer's first and last name, as separate elements either by using JAXB annotations or by representing the mapping in the TopLink OXM metadata format. The following procedures illustrate how to map values for customers' first names and last names

7.3.2.2.1 Mapping by Using JAXB Annotations Assuming the associated schema defines the following elements:

- `<"customer">` of the type `customer-type`, which itself is defined as `complexType`
- Sequential elements called `<"first-name">` and `<"last-name">`, both of the type `string`

You can use the `@XmlElement` annotation to map values for a customer's first and last names to the appropriate XML nodes. To do so:

1. Create the object and import `javax.xml.bind.annotation.*`:

```
package example;

import javax.xml.bind.annotation.*;
```

2. Declare the `Customer` class and use the `@XmlRootElement` annotation to make it the root element. Set the XML accessor type to `FIELD`:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
```

3. Define the `firstname` and `lastname` properties and annotate them with the `@XmlElement` annotation. Use the `name=` argument to customize the XML element name (if you do not explicitly set the name with `name=`, then the XML element will match the Java attribute name; for example, here the `<first-name>` element combination would be specified `<firstName> </firstName>` in XML).

```
   @XmlElement(name="first-name")
   private String firstName;

   @XmlElement(name="last-name")
   private String lastName;
```

The object should look like Example 7–16.

Example 7–16 Customer Object Mapping Values to a Simple Sequence

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
   @XmlElement(name="first-name")
   private String firstName;

   @XmlElement(name="last-name")
   private String lastName;

   ...
}
```

7.3.2.2.2 Defining the Mapping in OXM Metadata Format If you want to represent the mapping in the TopLink OXM metadata format, then use the XML tags defined in the `eclipselink-oxm.xml` file and populate them with the appropriate values, as shown in Example 7–17.

Example 7–17 Mapping Sequential Attributes in OXM Metadata Format

```
...
<java-type name="Customer">
   <xml-root-element name="customer"/>
   <java-attributes>
      <xml-element java-attribute="firstName" name="first-name"/>
      <xml-element java-attribute="lastName" name="last-name"/>
   </java-attributes>
</java-type>
```

...

7.3.2.3 Mapping a Value to a Text Node in a Subelement

You can map values from a Java object to text nodes that are nested as a subelement in the XML document by using JAXB annotations or by representing the mapping in the TopLink OXM metadata format. For example, if you want to populate `<first-name>` and `<last-name>` elements, which are subelements of a `<personal-info>` element under a `<customer>` root element, you could use the following procedures to achieve these mappings.

7.3.2.3.1 Mapping by Using JAXB Annotations Assume the associated schema defines the following elements:

- `<"customer">` of the type customer-type, which itself is defined as `complexType`
- `<personal-info>`
- Subelements of `<personal-info>` called `<"first-name">` and `<"last-name">`, both of the type String

You can use JAXB annotations to map values for a customer's first and last name to the appropriate XML subelement nodes. Because this example goes beyond a simple element name customization and actually introduces a new XML structure, it uses the TopLink `@XmlPath` annotation. To achieve this mapping:

1. Create the object and import `javax.xml.bind.annotation.*` and `org.eclipse.persistence.oxm.annotations.*`.

   ```
   package example;

   import javax.xml.bind.annotation.*;
   import org.eclipse.persistence.oxm.annotations.*;
   ```

2. Declare the `Customer` class and use the `@XmlRootElement` annotation to make it the root element. Set the XML accessor type to `FIELD`:

   ```
   @XmlRootElement
   @XmlAccessorType(XmlAccessType.FIELD)
   public class Customer {
   ```

3. Define the `firstName` and `lastName` properties.
4. Map the `firstName` and `lastName` properties to the subelements defined by the XML schema by inserting the `@XmlPath` annotation on the line immediately preceding the property declaration. For each annotation, define the mapping by specifying the appropriate XPath predicate:

   ```
   @XmlPath("personal-info/first-name/text()")
   private String firstName;

   @XmlPath("personal-info/last-name/text()")
   private String lastName;
   ```

The object should look like that shown in Example 7–18.

Example 7–18 Customer Object Mapping Properties to Subelements

```
package example;

import javax.xml.bind.annotation.*;
```

```
import org.eclipse.persistence.oxm.annotations.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
   @XmlPath("personal-info/first-name/text()")
   private String firstName;

   @XmlPath("personal-info/last-name/text()")
   private String lastName;

   ...
}
```

7.3.2.3.2 Defining the Mapping in OXM Metadata Format If you want to represent the mapping in the TopLink OXM metadata format, you need to use the XML tags defined in the `eclipselink-oxm.xml` file and populate them with the appropriate values, as shown in Example 7–19.

Example 7–19 Mapping Attributes as Subelements in OXM Metadata Format

```
...
<java-type name="Customer">
   <xml-root-element name="customer"/>
   <java-attributes>
      <xml-element java-attribute="firstName" xml-path="personal-info/first-name/text()"/>
      <xml-element java-attribute="lastName" xml-path="personal-info/last-name/text()"/>
   </java-attributes>
</java-type>
...
```

7.3.2.4 Mapping Values to a Text Node by Position

When multiple nodes have the same name, map their values from the Java object by specifying their position in the XML document. Do this by using mapping the values to the *position* of the attribute rather than the attribute's name. You can do this either by using JAXB annotations or by representing the mapping in the TopLink OXM metadata format. In the following example, XML contains two `<name>` elements; the first occurrence of `name` should represent the customer's first name and the second occurrence should represent the customer's last name.

Assume an XML schema defines the following attributes:

- `<customer>` of the type customer-type, which itself is specified as a `complexType`
- `<name>` of the type `String`

This example uses the JAXB `@XmlPath` annotation to map a customer's first and last names to the appropriate `<name>` element. It also uses the `@XmlType(propOrder)` annotation to ensure that the elements are always in the proper positions. To achieve this mapping:

1. Create the object and import `javax.xml.bind.annotation.*` and `org.eclipse.persistence.oxm.annotations.XmlPath`.

   ```
   package example;

   import javax.xml.bind.annotation.*;
   import org.eclipse.persistence.oxm.annotations.XmlPath;
   ```

2. Declare the `Customer` class and insert the `@XmlType(propOrder)` annotation with the arguments `"firstName"` followed by `"lastName"`. Insert the `@XmlRootElement` annotation to make `Customer` the root element and set the XML accessor type to `FIELD`:

```
@XmlRootElement
@XmlType(propOrder={"firstName", "lastName"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
```

3. Define the properties `firstName` and `lastName` with the type `String`.

4. Map the properties `firstName` and `lastName` to the appropriate position in the XML document by inserting the `@XmlPath` annotation with the appropriate XPath predicates.

```
@XmlPath("name[1]/text()")
private String firstName;

@XmlPath("name[2]/text()")
private String lastName;
```

   The predicates, `"name[1]/text()"` and `"name[2]/text()"` indicate the `<name>` element to which that specific property will be mapped; for example, `"name[1]/text"` will map the `firstName` property to the first `<name>` element.

The object should look like that shown in Example 7–20.

Example 7–20 Customer Object Mapping Values by Position

```
package example;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlPath;

@XmlRootElement
@XmlType(propOrder={"firstName", "lastName"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlPath("name[1]/text()")
    private String firstName;

    @XmlPath("name[2]/text()")
    private String lastName;

    ...
}
```

For more information on using XPath predicates, see Section 7.5, "Using XPath Predicates for Mapping".

7.4 Main Tasks for Using XML Metadata Representation to Override JAXB Annotations

In addition to using Java annotations, TopLink provides an XML mapping configuration file called `eclipselink-oxm.xml` that you can use in place of or to override JAXB annotations in the source with an XML representation of the metadata.

In addition to allowing all of the standard JAXB mapping capabilities, it also includes advanced mapping types and options.

An XML metadata representation is useful when:

- You cannot modify the domain model because, for example, it comes from a third party.
- You do not want to introduce compilation dependencies on JAXB APIs (if you are using a version of Java that predates Java SE 6).
- You want to apply multiple JAXB mappings to a domain model (you are limited to one representation with annotations).
- Your object model already contains so many annotations from other technologies that adding more would make the class unreadable.

Use the `eclipselink-oxm.xml` configuration file to override JAXB annotations by performing the following tasks:

- Task 1: Define Advanced Mappings in the XML
- Task 2: Configure Usage in JAXBContext
- Task 3: Specify MOXy as the JAXB Implementation

> **Caution:** While using this mapping file enables many advanced features, it might prevent you from porting it to other JAXB implementations.

7.4.1 Task 1: Define Advanced Mappings in the XML

First, update the XML mapping file to expose the `eclipselink_oxm_2_3.xsd` schema. Example 7–21 shows how to modify the `<xml-bindings>` element in the mapping file to point to the correct namespace and optimize the schema. Each Java package can have one mapping file.

Example 7–21 Updating XML Binding Information in the Mapping File

```
<?xml version="1.0"?>
<xml-bindings
       xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/oxm
http://www.eclipse.org/eclipselink/xsds/eclipselink_oxm_2_3.xsd"
       version="2.3">
</xml-bindings>
```

7.4.2 Task 2: Configure Usage in JAXBContext

Next, pass the mapping file to the `JAXBContext` class in your object:

1. Specify the externalized metadata by inserting this code:

```
Map<String, Source> metadata = new HashMap<String,Source>();
metadata.put("example.order", new StreamSource("order-metadata.xml"));
metadata.put("example.customer", new StreamSource("customer-metadata.xml"));
```

2. Create the properties object to pass to `JAXBContext`. For this example:

```
Map<String,Object> properties = new HashMap<String,Object>();
properties.put(JAXBContextFactory.ECLIPSELINK_OXM_XML_KEY, metadata);
```

3. Create `JAXBContext`. For example:

```
JAXBContext.newInstance("example.order:example.customer", aClassLoader,
properties);
```

7.4.3 Task 3: Specify MOXy as the JAXB Implementation

You must use MOXy as your JAXB implementation. To do so, do the following:

1. Open a `jaxb.properties` file and add the following line:

```
javax.xml.bind.context.factory=org.eclipse.persistence.jaxb.JAXBContextFactory
```

2. Copy the `jaxb.properties` file to the package that contains your domain classes.

7.5 Using XPath Predicates for Mapping

The TopLink MOXy API uses XPath predicates to define an expression that specifies the XML element's name. An XPath predicate is an expression that defines a specific object-to-XML mapping. As shown in previous examples, by default, JAXB will use the Java field name as the XML element name.

This section contains the following subsections:

- Section 7.5.1, "Understanding XPath Predicates"
- Section 7.5.2, "Main Tasks for Mapping Based on an Attribute Value"
- Section 7.5.3, "Self-Mappings"

7.5.1 Understanding XPath Predicates

As described previously, an XPath predicate is an expression that defines a specific object-to-XML mapping when standard annotations are not sufficient. For example, the following XML code shows a `<data>` element with two `<node>` subelements. If you want to create this mapping in a Java object, then specify an XPath predicate for each `<node>` subelement, for example `Node[2]` in the following Java:

```
<java-attributes>
   <xml-element java-attribute="node" xml-path="node[1]/ABC"/>
   <xml-element java-attribute="node" xml-path="node[2]/DEF"/>
</java-attributes>
```

This would match the second occurrence of the node element (`"DEF"`) in the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
   <node>ABC</node>
   <node>DEF</node>
</data>
```

Thus, by using the XPath predicate, you can use the same attribute name for a different attribute value.

In another example, if you wanted to map attributes based on position, you would follow the instructions described in Section 7.3.2.4, "Mapping Values to a Text Node by Position".

7.5.2 Main Tasks for Mapping Based on an Attribute Value

Beginning with TopLink MOXy 2.3, you can also map to an XML element based on an attribute value. In these tasks, you will annotate the JPA entity to render the XML document shown in Example 7–22. Note that all of the XML elements are named `node` but are differentiated by the value of their `name` attribute.

Example 7–22 JPA Entity

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
   <node name="first-name">Bob</node>
   <node name="last-name">Smith</node>
   <node name="address">
      <node name="street">123 A Street</node>
   </node>
   <node name="phone-number" type="work">555-1111</node>
   <node name="phone-number" type="cell">555-2222</node>
</node>
```

To attain this mapping, declare three classes, `Name`, `Address`, and `PhoneNumber` and then use an XPath in the form of *`element-name`*`[@`*`attribute-name`*`='`*`value`*`']` to map each Java field. To create entities from the Customer, Address, and PhoneNumber classes, perform the following tasks:

- Task 1: Create the Customer Class Entity
- Task 2: Create the Address Class Entity
- Task 3: Create the PhoneNumber Class Entity

7.5.2.1 Task 1: Create the Customer Class Entity

To create an entity from the `Customer` class:

1. Import the necessary JPA packages by adding the following code:

   ```
   import javax.xml.bind.annotation.*;

   import org.eclipse.persistence.oxm.annotations.XmlPath;
   ```

2. Declare the `Customer` class and use the `@XmlRootElement` annotation to make it the root element. Set the XML accessor type to `FIELD`:

   ```
   @XmlRootElement
   @XmlAccessorType(XmlAccessType.FIELD)
   public class Customer {
   ```

3. Declare these properties as local to the `Customer` class:

   - `firstName` (`String` type)
   - `lastName` (`String`)
   - `Address` (`Address`)

   For each property, set the Xpath predicate by preceding the property declaration with the annotation `@XmlPath(`*`element-name`*`[@`*`attribute-name`*`='`*`value`*`'])`. For example, for `firstName`, you would set the XPath predicate with this statement:

   ```
   @XmlPath("node[@name='first-name']/text()")
   ```

4. Also declare local to the `Customer` class the `phoneNumber` property as a `List<PhoneNumber>` type and assign it the value `new ArrayList<PhoneNumber>()`.

The `Customer` class should look like that shown in Example 7–23.

Example 7–23 Customer Object Mapping to an Attribute Value

```
package example;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlPath;

@XmlRootElement(name="node")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlPath("node[@name='first-name']/text()")
    private String firstName;

    @XmlPath("node[@name='last-name']/text()")
    private String lastName;

    @XmlPath("node[@name='address']")
    private Address address;

    @XmlPath("node[@name='phone-number']")
    private List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();

    ...
}
```

7.5.2.2 Task 2: Create the Address Class Entity

To create an entity from the `Address` class, do the following:

1. Import the necessary JPA packages by adding the following code:

   ```
   import javax.xml.bind.annotation.*;

   import org.eclipse.persistence.oxm.annotations.XmlPath;
   ```

2. Declare the `Address` class and set the XML accessor type to `FIELD`:

   ```
   @XmlAccessorType(XmlAccessType.FIELD)
   public class Address {
   ```

   This instance does not require the `@XmlRootElement` annotation as in Task 1: Create the Customer Class Entity because the `Address` class is not a root element in the XML document.

3. Declare the `String` property `street` local to the `Address` class. Set the XPath predicate by preceding the property declaration with the annotation `@XmlPath("node[@name='street']/text()")`.

The `Address` class should look like that shown in Example 7–24.

Example 7–24 Address Object Mapping to an Attribute Value

```
package example;
```

```
import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlPath;

@XmlAccessorType(XmlAccessType.FIELD)
public class Address {

    @XmlPath("node[@name='street']/text()")
    private String street;

    ...
}
```

7.5.2.3 Task 3: Create the PhoneNumber Class Entity

To create an entity from the `PhoneNumber` class:

1. Import the necessary JPA packages by adding the following code:

   ```
   import javax.xml.bind.annotation.*;

   import org.eclipse.persistence.oxm.annotations.XmlPath;
   ```

2. Declare the `PhoneNumber` class and use the `@XmlRootElement` annotation to make it the root element. Set the XML accessor type to `FIELD`:

   ```
   @XmlRootElement
   @XmlAccessorType(XmlAccessType.FIELD)
   public class Customer {
   ```

3. Create the `type` and `string` properties and define their mapping as attributes under the `PhoneNumber` root element by using the `@XmlAttribute`. annotation.

   ```
       @XmlAttribute
       private String type;

       @XmlValue
       private String number;
   ```

The `PhoneNumber` object should look like that shown in Example 7–25.

Example 7–25 PhoneNumber Object Mapping to an Attribute Value

```
package example;

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {

    @XmlAttribute
    private String type;

    @XmlValue
    private String number;

    ...
}
```

7.5.3 Self-Mappings

A self-mapping occurs on one-to-one mappings when you set the target object's XPath to "." (dot) so the data from the target object appears inside the source object's XML element. This exercise uses the example in Section 7.5.2, "Main Tasks for Mapping Based on an Attribute Value" to map the address information to appear directly under the customer element and not wrapped in its own element.

To create the self mapping:

1. Repeat Tasks 1 and 2 in Section 7.5.2.1, "Task 1: Create the Customer Class Entity".
2. Declare these properties local to the `Customer` class:
   - `firstName` (`String`)
   - `lastName` (`String`)
   - `Address` (`Address`)
3. For the `firstName` and `lastName` properties, set the `@XmlPath` annotation by preceding the property declaration with the annotation `@XmlPath(`*`element-name`*`[@`*`attribute-name`*`='`*`value`*`'])`. For example, for `firstName`, you would set the XPath predicate with this statement:

   ```
   @XmlPath("node[@name='first-name']/text()")
   ```

4. For the `address` property, set `@XmlPath` to "." (dot):

   ```
   @XmlPath(".")
   private Address address;
   ```

5. Also declare the `phoneNumber` property local to the `Customer` class. Declare it as a `List<PhoneNumber>` type and assign it the value `new ArrayList<PhoneNumber>()`.

The rendered XML for the `Customer` entity should look like that shown in Example 7–26.

Example 7–26 XML Node with Self-Mapped Address Element

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
   <node name="first-name">Bob</node>
   <node name="last-name">Smith</node>
   <node name="street">123 A Street</node>
   <node name="phone-number" type="work">555-1111</node>
   <node name="phone-number" type="cell">555-2222</node>
</node>
```

7.6 Using Dynamic JAXB/MOXy

Dynamic JAXB/MOXy allows you to bootstrap a `JAXBContext` class from a variety of metadata sources and use familiar JAXB APIs to marshal and unmarshal data, without requiring compiled domain classes. This is an enhancement over static JAXB, because now you can update the metadata without having to update and recompile the previously generated Java source code.

The benefits of using dynamic JAXB/MOXy entities are:

- Instead of using actual Java classes (for example, `Customer.class`, `Address.class`, and so on), the domain objects are subclasses of the `DynamicEntity` class.

- Dynamic entities offer a simple `get(propertyName)/set(propertyName propertyValue)` API to manipulate their data.
- Dynamic entities have an associated `DynamicType` class, which is generated in-memory, when the metadata is parsed.

7.6.1 Main Tasks for Using Dynamic JAXB/MOXy

The following Tasks demonstrate how to use dynamic JAXB/MOXy:

- Task 1: Bootstrap a Dynamic JAXBContext from an XML Schema
- Task 2: Create Dynamic Entities and Marshal Them to XML
- Task 3: Unmarshal the Dynamic Entities from XML

7.6.1.1 Task 1: Bootstrap a Dynamic JAXBContext from an XML Schema

Use the `DynamicJAXBContextFactory` class to create a dynamic `JAXBContext` object. Example 7–27 specifies the input stream and then bootstraps a `DynamicJAXBContext` class from the `customer.xsd` schema (Example 7–28) by using the `createContextFromXSD()` method.

Example 7–27 Specifying the Input Stream and Creating DynamicJAXBContext

```
import java.io.FileInputStream;

import org.eclipse.persistence.jaxb.dynamic.DynamicJAXBContext;
import org.eclipse.persistence.jaxb.dynamic.DynamicJAXBContextFactory;

public class Demo {

    public static void main(String[] args) throws Exception {
        FileInputStream xsdInputStream = new FileInputStream("src/example/customer.xsd");
        DynamicJAXBContext jaxbContext =
            DynamicJAXBContextFactory.createContextFromXSD(xsdInputStream, null, null, null);
```

The first parameter represents the XML schema itself and must be in one of the following forms: `java.io.InputStream`, `org.w3c.dom.Node`, or `javax.xml.transform.Source`.

7.6.1.1.1 The XML Schema Example 7–28 shows the `customer.xsd` schema that represents the metadata for the dynamic `JAXBContext` you are bootstrapping.

Example 7–28 Sample XML Schema Document

```
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://www.example.org"
   targetNamespace="http://www.example.org"
   elementFormDefault="qualified">

   <xsd:complexType name="address">
      <xsd:sequence>
         <xsd:element name="street" type="xsd:string" minOccurs="0"/>
         <xsd:element name="city" type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="customer">
      <xsd:complexType>
```

```
            <xsd:sequence>
               <xsd:element name="name" type="xsd:string" minOccurs="0"/>
               <xsd:element name="address" type="address" minOccurs="0"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

</xsd:schema>
```

7.6.1.1.2 Handling Schema Import/Includes

To bootstrap `DynamicJAXBContext` from an XML schema that contains imports of other schemas, you need to configure an `org.xml.sax.EntityResolver` to resolve the locations of the imported schemas and pass the `EntityResolver` to `DynamicJAXBContextFactory`.

Example 7–29 shows the schema document `customer.xsd` and Example 7–30 shows the schema document `address.xsd`. You can see that `customer.xsd` imports `address.xsd` by using the statement:

```
<xsd:import namespace="http://www.example.org/address"
schemaLocation="address.xsd"/>
```

Example 7–29 customer.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:add="http://www.example.org/address"
   xmlns="http://www.example.org/customer"
   targetNamespace="http://www.example.org/customer"
   elementFormDefault="qualified">

   <xsd:import namespace="http://www.example.org/address" schemaLocation="address.xsd"/>

   <xsd:element name="customer">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="name" type="xsd:string" minOccurs="0"/>
            <xsd:element name="address" type="add:address" minOccurs="0"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

</xsd:schema>
```

Example 7–30 address.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/address"
    targetNamespace="http://www.example.org/address"
    elementFormDefault="qualified">

    <xsd:complexType name="address">
        <xs:sequence>
            <xs:element name="street" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
        </xs:sequence>
    </xsd:complexType>

</xsd:schema>
```

7.6.1.1.3 Implementing and Passing EntityResolver If you want to bootstrap `DynamicJAXBContext` from the `customer.xsd` schema then pass an entity resolver. Do the following:

1. To resolve the locations of the imported schemas, implement `entityResolver` by supplying the code shown in Example 7–31.

Example 7–31 Implementing an EntityResolver

```
class MyEntityResolver implements EntityResolver {

    public InputSource resolveEntity(String publicId, String systemId) throws SAXException,
IOException {
      // Imported schemas are located in ext\appdata\xsd\

       // Grab only the filename part from the full path
      String filename = new File(systemId).getName();

      // Now prepend the correct path
      String correctedId = "ext/appdata/xsd/" + filename;

      InputSource is = new InputSource(ClassLoader.getSystemResourceAsStream(correctedId));
      is.setSystemId(correctedId);

      return is;
   }

}
```

2. After you implement `DynamicJAXBContext`, pass the `EntityResolver`, as shown in Example 7–32.

Example 7–32 Passing in the Entityresolver

```
FileInputStream xsdInputStream = new FileInputStream("src/example/customer.xsd");
DynamicJAXBContext jaxbContext =
    DynamicJAXBContextFactory.createContextFromXSD(xsdInputStream, new MyEntityResolver(), null,
null);
```

7.6.1.1.4 Error Handling You might see the following exception when importing another schema:

```
Internal Exception: org.xml.sax.SAXParseException: schema_reference.4: Failed to read schema
document '<imported-schema-name>', because 1) could not find the document; 2) the document could
not be read; 3) the root element of the document is not <xsd:schema>.
```

To work around this exception, disable the XJC schema correctness check by setting the `noCorrectnessCheck` Java property. You can set this property in one of two ways:

- From within the code, by adding this line:

  ```
  System.setProperty("com.sun.tools.xjc.api.impl.s2j.SchemaCompilerImpl.noCorrect
  nessCheck", "true")
  ```

- From the command line, by using this command:

  ```
  -Dcom.sun.tools.xjc.api.impl.s2j.SchemaCompilerImpl.noCorrectnessCheck=true
  ```

7.6.1.1.5 Specifying a Class Loader Use your application's current class loader as the `classLoader` parameter. This parameter verifies that specified classes exist before a new `DynamicType` is generated. In most cases you can pass `null` for this parameter and use `Thread.currentThread().getContextClassLoader()` method instead.

7.6.1.2 Task 2: Create Dynamic Entities and Marshal Them to XML

Use the `DynamicJAXBContext` class to create instances of a `DynamicEntity` object. The entity and property names correspond to the class and property names, in this case the customer and address, that would have been generated if you had used static JAXB.

Example 7–33 Creating the Dynamic Entity

```
DynamicEntity customer = jaxbContext.newDynamicEntity("org.example.Customer");
customer.set("name", "Jane Doe");

DynamicEntity address = jaxbContext.newDynamicEntity("org.example.Address");
address.set("street", "1 Any Street").set("city", "Any Town");
customer.set("address", address);
```

The marshaller obtained from the `DynamicJAXBContext` is a standard marshaller and can be used normally to marshal instances of `DynamicEntity`, as shown in Example 7–34.

Example 7–34 Standard Dynamic JAXB Marshaller

```
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(customer, System.out);
```

Example 7–35 show resultant XML document:

Example 7–35 Updated XML Document Showing <address> Element and Its Attributes

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="www.example.org">
   <name>Jane Doe</name>
   <address>
      <street>1 Any Street</street>
      <city>Any Town</city>
   </address>
</customer>
```

7.6.1.3 Task 3: Unmarshal the Dynamic Entities from XML

This task shows how to unmarshal from XML the dynamic entities you created in Task 2: Create Dynamic Entities and Marshal Them to XML. The XML in reference is shown in Example 7–35.

The Unmarshaller obtained from the `DynamicJAXBContext` object is a standard unmarshaller, and can be used to unmarshal instances of `DynamicEntity`, as shown in Example 7–36.

Example 7–36 Standard Dynamic JAXB Unmarshaller

```
FileInputStream xmlInputStream = new FileInputStream("src/example/dynamic/customer.xml");
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
DynamicEntity customer = (DynamicEntity) unmarshaller.unmarshal(xmlInputStream);
```

7.6.1.3.1 Get Data from the DynamicEntity Specify which data in the dynamic entity to obtain. Specify this value by using the `System.out.println()` method and passing in the entity name. `DynamicEntity` offers property-based data access; for example, `get("name")` instead of `getName()`:

```
System.out.println(customer.<String>get("name"));
```

7.6.1.3.2 Use DynamicType to Introspect Dynamic Entity Instances of `DynamicEntity` have a corresponding `DynamicType`, which you can use to introspect the `DynamicEntity` object, as shown in Example 7–37.

Example 7–37 Introspecting the DynamicEntity

```
DynamicType addressType = jaxbContext.getDynamicType("org.example.Address");

DynamicEntity address = customer.<DynamicEntity>get("address");
for(String propertyName: addressType.getPropertiesNames()) {
    System.out.println(address.get(propertyName));
}
```

7.7 Additional Resources

The following additional resources are available:

- Section 7.7.1, "Code Samples"
- Section 7.7.2, "Related Javadoc"

7.7.1 Code Samples

Numerous code samples and tutorials can be found at the EclipseLink MOXy wiki site:

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

7.7.2 Related Javadoc

The following Javadoc is available:

- Java Architecture for XML Binding (JAXB) Specification
- Mapping Objects to XML (MOXy) Specification

7.7.2.1 Java Architecture for XML Binding (JAXB) Specification

JAXB uses an extended set of annotations to define the binding rules for Java-to-XML mapping. These annotations are subclasses of the `javax.xml.bind.*` packages in the Oracle TopLink API. Javadoc is for these annotations is at:

`http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/package-summary.html`

7.7.2.2 Mapping Objects to XML (MOXy) Specification

MOXy supports all the standard JAXB annotations in the `javax.xml.bind.annotation` package:

`http://www.eclipse.org/eclipselink/api/2.3/org/eclipse/persistence/oxm/annotations/package-summary.html`

MOXy has its own extensions in the `org.eclipse.persistence.oxm.annotations` package:

http://www.eclipse.org/eclipselink/api/2.3/org/eclipse/persistence/oxm/annotations/package-summary.html

8

Testing TopLink JPA Outside a Container

With TopLink, you can use the persistence unit JAR to test your application outside the container (for instance, in applications for the Java Platform, Standard Edition (Java SE)).

This chapter includes the following sections:

8.1 Understanding JPA Deployment

When deploying outside of a container, use the `createEntityManagerFactory` method of the `javax.persistence.Persistence` class to create an entity manager factory. This method accepts a `Map` of properties and the name of the persistence unit. The properties that you pass to this method are combined with those specified in the `persistence.xml` file. They may be additional properties or they may override the value of a property that you specified previously in the `persistence.xml` file.

> **Tip:** This is a convenient way to set properties obtained from a program input, such as the command line.

8.1.1 Using an EntityManager

The `EntityManager` is the access point for persisting an entity bean loading it from the database. Normally, the JPA container manages interaction with the data source. However, if you are using a JTA data source for your JPA persistence unit, you can access the JDBC connection from the Java EE container's data source. You must include the connection information to the `persistence.xml` file because the managed data source is not available.

With Oracle TopLink, you also have access to the EclipseLink extensions to the `EntityManager`.

8.2 Configuring the persistence.xml File

The `persistence.xml` file is the deployment descriptor file for persistence using Java Persistence API (JPA). It specifies the persistence units and declares the managed persistence classes, the object/relation mapping, and the database connection details.

8.2.1 Main Tasks

This section includes the following tasks:

- Task 1: Use the persistence.xml File
- Task 2: Instantiate the EntityManagerFactory

8.2.1.1 Task 1: Use the persistence.xml File

Example 8–1 illustrates a `persistence.xml` file for a JavaSE configuration (that is, outside a container):

Example 8–1 A persistence.xml File Specifying JavaSE Configuration

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd"
version="1.0">
    <persistence-unit name="my-app" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="javax.persistence.jdbc.driver"
value="oracle.jdbc.OracleDriver"/>
            <property name="javax.persistence.jdbc.url"
value="jdbc:oracle:thin:@localhost:1521:orcl"/>
            <property name="javax.persistence.jdbc.user" value="scott"/>
            <property name="javax.persistence.jdbc.password" value="tiger"/>
        </properties>
    </persistence-unit>
</persistence>
```

8.2.1.2 Task 2: Instantiate the EntityManagerFactory

An `EntityManagerFactory` provides an efficient way to construct `EntityManager` instances for a database. You can instantiate the `EntityManagerFactory` for the application (illustrated in Example 8–1) by using:

```
Persistence.createEntityManagerFactory("my-app");
```

8.3 Using a Property Map

You can use a property map to override the default persistence properties.

8.3.1 Main Tasks

This section includes the following steps:

- Task 1: Configure the persistence.xml File
- Task 2: Configure the Bootstrapping API
- Task 3: Instantiate the EntityManagerFactory

8.3.1.1 Task 1: Configure the persistence.xml File

Example 8–2 illustrates a `persistence.xml` file that uses container deployment.

Example 8–2 A persistence.xml File Specifying JavaSE Configuration

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd"
version="1.0">
    <persistence-unit name="employee" transaction-type="RESOURCE_LOCAL">
        <non-jta-data-source>jdbc/MyDS</non-jta-data-source>
    </persistence-unit>
</persistence>
```

Note: There is no data source available when tested outside of a container.

8.3.1.2 Task 2: Configure the Bootstrapping API

To test the persistence unit shown in Example 8–2 outside the container, you must use the JavaSE bootstrapping API. Example 8–3 contains sample code that illustrates this bootstrapping:

Example 8–3 Sample Configuration

```
import static org.eclipse.persistence.config.PersistenceUnitProperties.*;

...

  Map properties = new HashMap();

  // Ensure RESOURCE_LOCAL transactions is used.
  properties.put(TRANSACTION_TYPE,
    PersistenceUnitTransactionType.RESOURCE_LOCAL.name());

  // Configure the internal connection pool
  properties.put(JDBC_DRIVER, "oracle.jdbc.OracleDriver");
  properties.put(JDBC_URL, "jdbc:oracle:thin:@localhost:1521:ORCL");
  properties.put(JDBC_USER, "scott");
  properties.put(JDBC_PASSWORD, "tiger");

  // Configure logging. FINE ensures all SQL is shown
  properties.put(LOGGING_LEVEL, "FINE");
  properties.put(LOGGING_TIMESTAMP, "false");
  properties.put(LOGGING_THREAD, "false");
  properties.put(LOGGING_SESSION, "false");

  // Ensure that no server-platform is configured
  properties.put(TARGET_SERVER, TargetServer.None);
```

8.3.1.3 Task 3: Instantiate the EntityManagerFactory

An `EntityManagerFactory` provides an efficient way to construct `EntityManager` instances for a database. You can instantiate the `EntityManagerFactory` for the application (illustrated in Example 8–3) by using:

```
Persistence.
createEntityManagerFactory("unitName", "Properties");
```

8.4 Additional Resources

For additional information on JPA deployment, see the following sections of the JPA Specification (http://jcp.org/en/jsr/detail?id=317):

- Section 7.2, "Bootstrapping in Java SE Environments"
- Chapter 7, "Container and Provider Contracts for Deployment and Bootstrapping"

8.4.1 Javadoc

For more information, see the following APIs in *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

- `PersistenceUnitProperties` class
- `EntityManagerFactory` interface
- `JpaEntityManager` interface

9 Enhancing TopLink Performance

This chapter describes the Oracle TopLink performance features and how to monitor and optimize TopLink-enabled applications.

This chapter contains the following sections:

- Section 9.1, "Performance Features"
- Section 9.2, "Using Tools to Monitor and Optimize TopLink-Enabled Applications"

9.1 Performance Features

Toplink includes a number of performance features that make it the industry's best performing and most scalable JPA implementation. These features include:

- Object Caching
- Querying
- Enhancing Mapping Performance
- Transactions
- Database

9.1.1 Object Caching

The TopLink cache is an in-memory repository that stores recently read or written objects based on class and primary key values. The cache helps improve performance by holding recently read or written objects and accessing them in-memory to minimize database access.

Caching allows you to:

- Set how long the cache lasts and the time of day; this is a process called cache invalidation.
- Configure cache types (`Weak`, `Soft`, `SoftCache`, `HardCache`, `Full`) on a per entity basis.
- Configure cache size on a per entity basis.
- Coordinate clustered caches.

9.1.1.1 Caching Annotations

TopLink defines these entity-caching annotations:

- `@Cache`

- `@TimeOfDay`
- `@ExistenceChecking`

TopLink also provides a number of persistence unit properties that you can specify to configure the TopLink cache (see "How to Use the Persistence Unit Properties for Caching" in the EclipseLink online documentation, at http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_Persistence_Unit_Properties_for_Caching). These properties might compliment or provide an alternative to using annotations.

9.1.1.2 Using the @Cache Annotation

TopLink uses identity maps to cache objects in order to enhance performance, as well as maintain object identity. You can control the cache and its behavior by using the `@Cache` annotation in your entity classes. Example 9–1 shows how to implement this annotation.

Example 9–1 Using the @Cache Annotation

```
@Entity
 @Table(name="EMPLOYEE")
 @Cache (
     type=CacheType.WEAK,
     isolated=false,
     expiry=600000,
     alwaysRefresh=true,
     disableHits=true,
     coordinationType=INVALIDATE_CHANGED_OBJECTS
     )
 public class Employee implements Serializable {
     ...
 }
```

For more information about object caching and using the `@Cache` annotation, see "Using EclipseLink JPA Extensions for Entity Caching" in the EclipseLink online documentation:

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_Entity_Caching

9.1.2 Querying

The scope of a query, the amount of data returned, and how that data is returned can all affect the performance of a TopLink-enabled application. The Toplink query mechanisms enhance query performance by providing these features:

- Read-Only Queries
- Join Fetching Feature
- Batch Reading
- Fetch Size
- Pagination
- Cache Usage

9.1.2.1 Read-Only Queries

TopLink uses the `eclipselink.read-only` hint, `QueryHint` (`@QueryHint`), to retrieve read-only results from a query. On non-transactional read operations, where the requested entity types are stored in the shared cache, you can request that the shared instance be returned instead of a detached copy.

For more information about read-only queries, see the documentation for the read-only hint:

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Read_Only

9.1.2.2 Join Fetching Feature

The Join Fetching feature enhances performance by enabling the joining and reading of the related objects in the same query as the source object. Enable Join Fetching by using the `@JoinFetch` annotation, as shown in Example 9–2. This example shows how the `@JoinFetch` annotation specifies the `Employee` field `managedEmployees`.

Example 9–2 Enabling the Join Fetching Feature

```
@Entity
 public class Employee implements Serializable {
     ...
     @OneToMany(cascade=ALL, mappedBy="owner")
     @JoinFetch(value=OUTER)
     public Collection<Employee> getManagedEmployees() {
         return managedEmployees;
     }
     ...
 }
```

For more information about Join Fetching, see "How to Use the @JoinFetch Annotation" in the EclipseLink online documentation:

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40JoinFetch_Annotation

9.1.2.3 Batch Reading

The `eclipselink.batch` hint supplies TopLink with batching information so subsequent queries of related objects can be optimized in batches instead of being retrieved one-by-one or in one large joined read. Batch reading is more efficient than joining because it avoids reading duplicate data. Batching is allowed on queries that have only a single object in their Select clause.

9.1.2.4 Fetch Size

When you have large queries that return a large number of objects, you can improve performance by reducing the number of database hits required to satisfy the selection criteria. To do this, use the `eclipselink.jdbc.fetch-size` hint. This hint specifies the number of rows that should be fetched from the database when more rows are required (depending on the JDBC driver support level). Most JDBC drivers default to a fetch size of 10, so if you are reading 1000 objects, then increasing the fetch size to 256 can significantly reduce the time required to fetch the query's results. The optimal fetch size is not always obvious. Usually, a fetch size of one half or one quarter of the total expected result size is optimal. Note that setting a fetch size too large or too small can decrease performance.

9.1.2.5 Pagination

Slow paging can result in significant application overhead; however, TopLink includes a variety of solutions for improving paging results. For example, you can:

- Configure the first and maximum number of rows to retrieve when executing a query.
- Perform a query on the database for all of the ID values that match the criteria and then use these values to retrieve specific sets.
- Configure TopLink to return a `ScrollableCursor` object from a query by using query hints. This returns a database cursor on the query's result set and allows the client to scroll through the results page by page.

For information about improving paging performance, see "How to use EclipseLink Pagination" in the EclipseLink online documentation, at:

`http://wiki.eclipse.org/EclipseLink/Examples/JPA/Pagination#How_to_use_EclipseLink_Pagination`

9.1.2.6 Cache Usage

TopLink uses a shared cache mechanism that is scoped to the entire persistence unit. When operations are completed in a particular persistence context, the results are merged back into the shared cache so that other persistence contexts can use them. This happens regardless of whether the entity manager and persistence context are created in the Java SE or Java EE platform. Any entity persisted or removed using the entity manager will always be consistent with the cache.

You can specify how the query should interact with the TopLink cache by using the `eclipselink.cache-usage` hint. For more information, see "Cache Usage" in the EclipseLink online documentation:

`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Cache_Usage`

9.1.3 Enhancing Mapping Performance

Mapping performance is enhanced by these features:

- Indirection ("Lazy Loading")
- Read-Only Classes
- Weaving

9.1.3.1 Indirection ("Lazy Loading")

By default, when TopLink retrieves a persistent object, it retrieves all of the dependent objects to which it refers. When you configure indirection (also known as lazy loading, lazy reading, and just-in-time reading) for an attribute mapped with a relationship mapping, TopLink uses an indirection object as a placeholder for the referenced object. TopLink defers reading the dependent object until you access that specific attribute. This can result in a significant performance improvement, especially if the application is interested only in the contents of the retrieved object, rather than the objects to which it is related.

TopLink supports different types of indirection, including: value holder indirection, transparent indirect container indirection, and proxy indirection.

9.1.3.2 Read-Only Classes

When you declare a class read-only, clones of that class are neither created nor merged, greatly improving performance. You can declare a class as read-only within the context of a unit of work by using the `addReadOnlyClass()` method by using one of the following techniques:

- To configure a read-only class for a single unit of work, specify that class as the argument to the `addReadOnlyClass()` method:

  ```
  myUnitofWork.addReadOnlyClass(B.class);
  ```

- To configure multiple classes as read-only, add them to a vector and specify that vector as the argument to the `addReadOnlyClass()` method:

  ```
  myUnitOfWork.addReadOnlyClasses(myVectorOfClasses);
  ```

For more information about using read-only classes to enhance performance, see "Declaring Read-Only Classes" in the EclipseLink online documentation:

http://wiki.eclipse.org/Using_Advanced_Unit_of_Work_API_%28ELUG%29#Declaring_Read-Only_Classes

9.1.3.3 Weaving

Weaving is a technique of manipulating the byte code of compiled Java classes. The TopLink JPA persistence provider uses weaving to enhance both JPA entities and Plain Old Java Object (POJO) classes for such things as lazy loading, change tracking, fetch groups, and internal optimizations.

Weaving can be performed either dynamically at runtime, when entities are loaded, or statically at compile time by post-processing the entity `.class` files. By default, TopLink uses dynamic weaving whenever possible. This includes inside a Java EE 5 or Java EE 6 application server and in the Java SE platform when the TopLink agent is configured. Dynamic weaving is recommended because it is easy to configure and does not require any changes to a project's build process.

For information about how to use weaving to enhance application performance, see "Weaving" in the EclipseLink online documentation:

http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Performance/Weaving

9.1.4 Transactions

You can optimize performance during data transactions by using change tracking. Change tracking allows you to tune the way TopLink detects changes that occur during a transaction. You should choose the strategy based on the usage and data modification patterns of the entity type because different types may have different access patterns and hence different settings, and so on.

Enable change tracking by using the `@ChangeTracking` annotation, as shown in Example 9–3.

Example 9–3 Enabling Change Tracking

```
@Entity
@Table(name="EMPLOYEE")
@ChangeTracking(OBJECT) (
public class Employee implements Serializable {
    ...
}
```

For more information about change tracking, see "Using EclipseLink JPA Extensions for Tracking Changes" in the EclipseLink online documentation, at:

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_Tracking_Changes

9.1.5 Database

Database performance features in TopLink include:

- Connection Pooling
- Parameterized SQL and Statement Caching
- Batch Writing

9.1.5.1 Connection Pooling

Establishing a connection to a data source can be time-consuming, so reusing such connections in a connection pool can improve performance. TopLink uses connection pools to manage and share the connections used by server and client sessions. This feature reduces the number of connections required and allows your application to support many clients.

By default, TopLink sessions use internal connection pools. These pools allow you to optimize the creation of read connections for applications that read data only to display it and only infrequently modify data. They also allow you to use TopLink Workbench to configure the default (write) and read connection pools and to create additional connection pools for object identity or any other purpose.

In addition to internal connection pools, you can also configure TopLink to use any of these types of connection pools:

- External connection pools: You must use this type of connection pool to integrate with external transaction controller.
- Default (write) and read connection pools.
- Sequence connection pools: Use these pools when your application requires table sequencing (that is, nonnative sequencing) and you are using an external transaction controller.
- Application-specific connection pools: These are connection pools that you can create and use for any application purpose, provided you are using internal TopLink connection pools in a session.

For more information about using connection pools with TopLink, see the following topics in the EclipseLink online documentation:

- "Connection Pools":

  http://wiki.eclipse.org/Introduction_to_Data_Access_%28ELUG%29#Connection_Pools

- "Introduction to the Internal Connection Pool Creation":

  http://wiki.eclipse.org/Creating_an_Internal_Connection_Pool_%28ELUG%29#Introduction_to_the_Internal_Connection_Pool_Creation

9.1.5.2 Parameterized SQL and Statement Caching

Parameterized SQL can prevent the overall length of a SQL query from exceeding the statement length limit that your JDBC driver or database server imposes. Using parameterized SQL along with prepared statement caching can improve performance by reducing the number of times the database SQL engine parses and prepares SQL for a frequently called query.

By default, TopLink enables parameterized SQL, but not prepared statement caching. You should enable statement caching either in TopLink when using an internal connection pool or in the data source when using an external connection pool and you want to specify a statement cache size appropriate for your application.

To enable parameterized SQL, add this line to the `persistence.xml` file that is in the same path as your domain classes:

```
<property name="eclipselink.jdbc.bind-parameters" value="true"/>
```

To disable parameterized SQL, change `value=` to `false`.

For more information about using parameterized SQL and statement caching, see "How to Use Parameterized SQL (Parameter Binding) and Prepared Statement Caching for Optimization" in the EclipseLink online documentation:

`http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#How_to_Use_Parameterized_SQL_.28Parameter_Binding.29_and_Prepared_Statement_Caching_for_Optimization`

9.1.5.3 Batch Writing

Batch writing helps optimize transactions with multiple write operations. Batch writing is enabled by using the TopLink JDBC extension `batch-writing`. You set one of the following parameter this property into the session at deployment time:

- `JDBC`: Use JDBC batch writing.
- `Buffered`: Do not use either JDBC batch writing nor native platform batch writing.
- `Oracle-JDBC`: Use both JDBC batch writing and Oracle native platform batch writing and use `OracleJDBC` in your property map.
- `None`: Disable batch writing.

For more information about batch writing, see "How to Use EclipseLink JPA Extensions for JDBC Connection Communication" in the EclipseLink online documentation:

`http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_JDBC`

9.2 Using Tools to Monitor and Optimize TopLink-Enabled Applications

The most important challenge to performance tuning is knowing what to optimize. To improve the performance of your application, identify the areas of your application that do not operate at peak efficiency.

Oracle TopLink provides a diverse set of features to measure and optimize application performance. You can enable or disable most features in the descriptors or session, making any resulting performance gains global.Performance considerations are present at every step of the development cycle. Although this implies an awareness of

performance issues in your design and implementation, it does not mean that you should expect to achieve the best possible performance in your first pass.

For example, if optimization complicates the design, leave it until the final development phase. You should still plan for these optimizations from your first iteration, to make them easier to integrate later.

9.2.1 Main Tasks

The most important concept associated with tuning your TopLink application is the idea of an iterative approach. The most effective way to tune your application is to do the following tasks:

- Task 1: Measure TopLink Performance with the TopLink Profiler
- Task 2: Identify Sources of Application Performance Problems
- Task 3: Modify Poorly Performing Application Components
- Task 4: Measure Performance Again

9.2.2 Task 1: Measure TopLink Performance with the TopLink Profiler

The TopLink performance profiler helps you identify performance problems by logging performance statistics for every executed query in a given session.

The TopLink performance profiler logs to the log file the information in Table 9–1.

Table 9–1 Information Logged by the TopLink Performance Profiler

Information Logged	Description
Query Class	Query class name
Domain Class	Domain class name
Total Time	Total execution time of the query, including any nested queries (in milliseconds)
Local Time	Execution time of the query, excluding any nested queries (in milliseconds)
Number of Objects	The total number of objects affected
Number of Objects Handled per Second	How many objects were handled per second of transaction time
Logging	the amount of time spent printing logging messages (in milliseconds)
SQL Prepare	The amount of time spent preparing the SQL script (in milliseconds)
SQL Execute	The amount of time spent executing the SQL script (in milliseconds)
Row Fetch	The amount of time spent fetching rows from the database (in milliseconds)
Cache	The amount of time spent searching or updating the object cache (in milliseconds)
Object Build	The amount of time spent building the domain object (in milliseconds)
Query Prepare	The amount of time spent to prepare the query prior to execution (in milliseconds)

Table 9–1 (Cont.) Information Logged by the TopLink Performance Profiler

Information Logged	Description
SQL Generation	The amount of time spent to generate the SQL script before it is sent to the database (in milliseconds)

9.2.2.1 Enabling the TopLink Profiler

The TopLink performance profiler is an instance of the `org.eclipse.persistence.tools.profiler.PerformanceProfiler` class. To enable it, add the following line to the `persistence.xml` file:

```
<property name="eclipselink.profiler" value="PerformanceProfiler.logProfiler"/>
```

In addition to enabling the TopLink profiler, `PerformanceProfiler` class public API also provides the functionality described in Table 9–2.

Table 9–2 Additional PerformanceProfiler Functionality

To...	Use...
Disable the profiler	`dontLogProfile`
Organize the profiler log into a summary of all the individual operation profiles, including these operation statistics: ■ Operations that were profiled ■ Shortest profiling operation ■ Total time of all the operations ■ Number of objects returned by profiled queries ■ Total time that was spent in each kind of operation that was profiled	`logProfileSummary`
Organize the profiler log into a summary of all the individual operation profiles by query	`logProfileSummaryByQuery`
Organize the profiler log into a summary of all the individual operation profiles by class.	`logProfileSummaryByClass`

9.2.2.2 Accessing and Interpreting Profiler Results

You can see profiling results by opening the profile log in a text reader, such as Notepad.

The profiler output file indicates the health of a TopLink-enabled application.

Example 9–4 shows an example of the TopLink profiler output.

Example 9–4 Performance Profiler Output

```
Begin Profile of{
ReadAllQuery(com.demos.employee.domain.Employee)
Profile(ReadAllQuery,# of obj=12, time=139923809,sql execute=21723809,
prepare=49523809, row fetch=39023809, time/obj=11623809,obj/sec=8)
} End Profile
```

Example 9–4 shows the following information about the query:

- `ReadAllQuery(com.demos.employee.domain.Employee)`: specific query profiled, and its arguments
- `Profile(ReadAllQuery`: start of the profile and the type of query

- `# of obj=12`: number of objects involved in the query
- `time=139923809`: total execution time of the query (in milliseconds)
- `sql execute=21723809`: total time spent executing the SQL statement
- `prepare=49523809`: total time spent preparing the SQL statement
- `row fetch=39023809`: total time spent fetching rows from the database
- `time/obj=116123809`: number of nanoseconds spent on each object
- `obj/sec=8`: number of objects handled per second

9.2.3 Task 2: Identify Sources of Application Performance Problems

Areas of the application where performance problems could occur include the following:

- Identifying General Performance Optimization
- Schema
- Mappings and Descriptors
- Sessions
- Cache
- Data Access
- Queries
- Unit of Work
- Application Server and Database Optimization

Task 3: Modify Poorly Performing Application Components provides some guidelines for dealing with problems in each of these areas.

9.2.4 Task 3: Modify Poorly Performing Application Components

For each potential source of application performance problems listed in Section 9.2.3, "Task 2: Identify Sources of Application Performance Problems", you can try specific workarounds.

9.2.4.1 Identifying General Performance Optimizations

Avoid overriding TopLink default behavior unless your application requires it. Some of these defaults are suitable for a development environment; you should change these defaults to suit your production environment. See "Optimizing for a Production Environment" in the EclipseLink online documentation:

`http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_for_a_Production_Environment`.

Use the Workbench rather than manual coding. These tools are not only easy to use, but the default configuration they export to deployment XML (and the code it generates, if required) represents best practices optimized for most applications.

9.2.4.2 Schema

Optimization is an important consideration when you design your database schema and object model. Most performance issues occur when the object model or database schema is too complex, because this can make the database slow and difficult to query.

This is most likely to happen if you derive your database schema directly from a complex object model.

To optimize performance, design the object model and database schema together. However, allow each model to be designed optimally: do not require a direct one-to-one correlation between the two.

For information about designing schema, see "Optimizing Schema", in the EclipseLink online documentation (http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_Schema). This document includes four schema optimization scenarios that will help you design schema that provides the desired performance.

9.2.4.3 Mappings and Descriptors

If you find performance bottlenecks in your mapping and descriptors, then try these workarounds:

- Always use indirection (lazy loading). It is not only critical in optimizing database access, but also allows TopLink to make several other optimizations including optimizing its cache access and unit of work processing. See "Configuring Indirection (Lazy Loading)" in the EclipseLink online documentation:

  http://wiki.eclipse.org/Configuring_a_Mapping_%28ELUG%29#Configuring_Indirection_.28Lazy_Loading.29

- Avoid using method access in your TopLink mappings, especially if you have expensive or potentially dangerous side-effect code in your `get` or `set` methods; use the default direct attribute access instead. See "Configuring Method or Direct Field Accessing at the Mapping Level" in the EclipseLink online documentation:

  http://wiki.eclipse.org/Configuring_a_Descriptor_%28ELUG%29#Configuring_Cache_Existence_Checking_at_the_Descriptor_Level

- Avoid using the existence checking option `checkCacheThenDatabase` on descriptors, unless required by the application. The default existence checking behavior offers better performance. See "Configuring Cache Existence Checking at the Descriptor Level" in the EclipseLink online documentation:

  http://wiki.eclipse.org/Configuring_a_Mapping_%28ELUG%29#Configuring_Method_or_Direct_Field_Accessing_at_the_Mapping_Level

- Avoid expensive initialization in the default constructor that TopLink uses to instantiate objects. Instead, use lazy initialization or use a TopLink instantiation policy to configure the descriptor to use a different constructor. See Configuring Instantiation Policy" in the EclipseLink online documentation:

  http://wiki.eclipse.org/Configuring_a_Descriptor_%28ELUG%29#Configuring_Instantiation_Policy

9.2.4.4 Sessions

If you suspect that session performance is hindering your application, try these workarounds:

- Use a server session in a server environment instead of a database session.
- Use the TopLink client session instead of a remote session. A client session is appropriate for most multiuser Java EE application server environments.
- Do not pool client sessions. Pooling sessions offers no performance gains.

- Increase the size of your session read and write connection pools to the desired number of concurrent threads (for example, 50). You can configure this in TopLink when you are using an internal connection pool or in the data source when you are using an external connection pool.

For a list of additional resources, see "Optimizing Sessions" in the EclipseLink online documentation:

http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_Sessions

9.2.4.5 Cache

You can often improve cache performance by implementing cache coordination. Cache coordination allows multiple, possibly distributed instances of a session to broadcast object changes among each other so that each session's cache can be kept up-to-date. For detailed information about optimizing cache behavior, see "Optimizing Cache" in the EclipseLink online documentation, at:

http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_Cache

9.2.4.6 Data Access

Depending on the type of data source your application accesses, TopLink offers a variety of `Login` options that you can use to tune the performance of low level data read and write operations. For optimizing higher-level data read and write operations, see "Optimizing Data Access" (in the EclipseLink online documentation at http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_Data_Access).

This document offers several techniques to improve data access performance for your application. These techniques show you how to:

- Optimize JDBC driver properties.
- Optimize data format.
- Use batch writing for optimization.
- Use outer-join reading with inherited subclasses.
- Use parameterized SQL (parameter binding) and prepared statement caching for optimization.

9.2.4.7 Queries

TopLink provides an extensive query API for reading, writing, and updating data. For information about improving query performance, see "Optimizing Queries" (in the EclipseLink online documentation at http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_Queries).

This document offers several techniques to improve query performance for your application. These techniques show you how to:

- Use parameterized SQL and prepared statement caching for optimization.
- Use named queries for optimization.
- Use batch and join reading for optimization.
- Use partial object queries and fetch groups for optimization.
- Use read-only queries for optimization.

- Use JDBC fetch size for optimization.
- Use cursored streams and scrollable cursors for optimization.
- Use result set pagination for optimization.

It also includes links to read and write optimization examples.

9.2.4.8 Unit of Work

To obtain optimal performance when using a unit of work, consider the following tips:

- Register objects with a unit of work only if objects are eligible for change. If you register objects that will not change, then the unit of work needlessly clones and processes those objects.
- Avoid the performance cost of existence checking when you are registering a new or existing object. For more information, see "How to Use Registration and Existence Checking" in the EclipseLink online documentation:

  `http://wiki.eclipse.org/Using_Advanced_Unit_of_Work_API_%28ELUG%29#How_to_Use_Registration_and_Existence_Checking`
- Avoid the performance cost of change set calculation on a class you know will not change by telling the unit of work that the class is read-only. For more information, see "Declaring Read-Only Classes" in the EclipseLink online documentation:

  `http://wiki.eclipse.org/Using_Advanced_Unit_of_Work_API_%28ELUG%29#Declaring_Read-Only_Classes`
- Avoid the performance cost of change set calculation on an object read by a `ReadAllQuery` in a unit of work that you do not intend to change by unregistering the object. For more information, see "How to Unregister Working Clones" in the EclipseLink online documentation:

  `http://wiki.eclipse.org/Using_Advanced_Unit_of_Work_API_%28ELUG%29#How_to_Unregister_Working_Clones`
- Before using conforming queries, ensure that it is necessary. For alternatives, see "Using Conforming Queries and Descriptors" in the EclipseLink online documentation:

  `http://wiki.eclipse.org/Using_Advanced_Unit_of_Work_API_%28ELUG%29#Using_Conforming_Queries_and_Descriptors`
- Enable weaving and change tracking to greatly improve transactional performance. For more information, see "Optimizing Using Weaving" in the EclipseLink online documentation:

  `http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_Using_Weaving`

If your performance measurements show that you have a performance problem during unit of work commit transaction, then consider using object-level or attribute-level change tracking, depending on the type of objects involved and how they typically change. For more information, see "Unit of Work and Change Policy" in the EclipseLink online documentation:

`http://wiki.eclipse.org/Introduction_to_EclipseLink_Transactions_%28ELUG%29#Unit_of_Work_and_Change_Policy`

9.2.4.9 Application Server and Database Optimization

To optimize the application server and database performance, consider these techniques:

- Configuring your application server and database correctly can have a big impact on performance and scalability. Ensure that you correctly optimize these key components of your application in addition to your TopLink application and persistence.
- For your application or Java EE platform, ensure that your memory, thread pool, and connection pool sizes are sufficient for your server's expected load, and that your JVM has been configured optimally.
- Ensure that your database has been configured correctly for optimal performance and its expected load.

9.2.5 Task 4: Measure Performance Again

Finally, after identifying possible performance bottlenecks and taking some action on them, rerun your application, again with the profiler enabled (see Section 9.2.2.1, "Enabling the TopLink Profiler"). Review the results and, if more action is required, then follow the procedures outlined in Section 9.2.4, "Task 3: Modify Poorly Performing Application Components".

10

Migrating From Hibernate to TopLink

This chapter describes how to migrate applications from using Hibernate JPA annotations and its native and proprietary API to using TopLink JPA. The migration involves converting Hibernate annotations to TopLink's native annotations, and converting native Hibernate API to TopLink JPA in the application code. Standard JPA annotations and API are left unchanged.

This chapter contains the following sections:

10.1 Understanding Hibernate

Hibernate is an object-relational mapping tool for Java environments. It provides a framework for mapping Java objects to relational database artifacts, and Java data types to SQL data types. It also provides the ability to query the database and retrieve data.

For more information about Hibernate, see `http://www.hibernate.org`.

Motivations for Migrating

Reasons why you would want to migrate from Hibernate to TopLink include:

- **Performance and scalability**: TopLink's caching architecture allows you to minimize object creation and share instances. TopLink's caching supports single-node and clustered deployments.
- **Support for leading relation databases**: TopLink continues to support all leading relational databases with extensions specific to each. TopLink is also the best ORM solution for the Oracle database.
- **A comprehensive persistence solution**: While TopLink offers industry leading object-relational support, TopLink also uses its core mapping functionality to deliver Object-XML (JAXB), Service Data Object (SDO), and Database Web Services. Depending on your requirements you can use one or more of the persistence services based on the same core persistence engine.
- **JPA Support**: As the JPA 1.0 specification co-leads, Oracle and the TopLink/EclipseLink team have been focused on delivering a JPA-compliant solution with supporting integration with JDeveloper, ADF, Spring, and the Eclipse IDE (Dali project). Oracle has delivered the JPA 1.0 reference implementation and EclipseLink delivers the JPA 2.0 reference implementation.

Oracle is focussed on standards-based development, while still offering many advanced capabilities.

10.2 Main Tasks

Complete these tasks to migrate an application that uses Hibernate as its persistence provider to Oracle TopLink.

- Task 1: Convert the Hibernate Entity Annotation
- Task 2: Convert the Hibernate Custom Sequence Generator Annotation
- Task 3: Convert Hibernate Mapping Annotations
- Task 4: Modify the persistence.xml File
- Task 5: Convert Hibernate API to EclipseLink API

10.2.1 Task 1: Convert the Hibernate Entity Annotation

The Hibernate entity annotation, defined by the `org.hibernate.annotations.Entity` class, adds additional metadata beyond what is defined by the JPA standard `@Entity` annotation.

Example 10–1 illustrates a sample Hibernate entity annotation. The example uses the `selectBeforeUpdate`, `dynamicInsert`, `dynamicUpdate`, `optimisticLock`, and `polymophism` attributes. Note that the Hibernate entity annotation also defines `mutable` and `persister` attributes which are not used in this example.

Example 10–1 Sample Hibernate Entity Annotation

```
@org.hibernate.annotations.Entity(
     selectBeforeUpdate = true,
     dynamicInsert = true,
     dynamicUpdate = true,
     optimisticLock = OptimisticLockType.ALL,
     polymorphism = PolymorphismType.EXPLICIT)
```

The following sections describe how TopLink treats selects, dynamic updates and inserts, locks, and polymorphism. For more information, see "EclipseLink/Examples/JPA/Migration/Hibernate/V3Annotations" in the Eclipselink documentation.

http://wiki.eclipse.org/EclipseLink/Examples/JPA/Migration/Hibernate/V3Annotations

10.2.1.1 Convert the Select Before Update, Dynamic Insert and Update Attributes

In Hibernate, the `selectBeforeUpdate` attribute specifies that Hibernate should never perform an SQL `UPDATE` unless it is certain that an object is actually modified. The `dynamicInsert` attribute specifies that `INSERT SQL` should be generated at runtime and contain only the columns whose values are not null. The `dynamicUpdate` attribute specifies that `UPDATE SQL` should be generated at runtime and can contain only those columns whose values have changed.

By default, TopLink will always insert all mapped columns and will update only the columns that have changed. If alternate operations are required, then the queries used for these operations can be customized by using Java code, SQL, or stored procedures.

10.2.1.2 Convert the Optimistic Lock Attribute

In Hibernate, the `optimisticLock` attribute determines the optimistic locking strategy.

TopLink's optimistic locking functionality supports all of the Hibernate locking types and more. Table 10–1 translates locking types from Hibernate's `@Entity(optimisticLock)` attributes into TopLink locking policies. These policies can be configured either with the TopLink `@OptimisticLocking` annotation or in the TopLink `orm.xml` file. For more information, see "Using EclipseLink JPA Extensions for Optimistic Locking" in the EclipseLink documentation.

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_Optimistic_Locking

Table 10–1 Transforming Hibernate's OptimisticLock to TopLink's OptimisticLocking

Hibernate's OptimisticLock Type	Description	EclipseLink OptimisticLocking
`NONE`	No optimistic locking	EclipseLink defaults to no optimistic locking
`VERSION`	Use a column version	Use the JPA `@Version` annotation or EclipseLink annotation: `@OptimisticLocking(type = OptimisticLockingType.VERSION_COLUMN)`
`DIRTY`	Changed columns are compared	Use the JPA `@Version` annotation or the EclipseLink annotation: `@OptimisticLocking(type = OptimisticLockingType.CHANGED_COLUMNS)`
`ALL`	All columns are compared	Use the EclipseLink annotation: `@OptimisticLocking(type = OptimisticLockingType.ALL_COLUMNS)`

Additionally, TopLink allows you to compare a specific set of selected columns using the `OptimisticLockingType.SELECTED_COLUMNS` annotation. This allows you to select the critical columns that should be compared if the `CHANGED` or `ALL` strategies do not meet your needs.

10.2.2 Task 2: Convert the Hibernate Custom Sequence Generator Annotation

In Hibernate, the `@GeneratedValue` annotation defines the identifier generation strategy. The `@GenericGenerator` allows you to define a Hibernate-specific ID generator. Example 10–2 illustrates a custom generator for sequence values.

Example 10–2 Custom Generator for Sequence Values

```
...
@Id
   @GeneratedValue(generator = "system-uuid")
   @GenericGenerator(name = "system-uuid", strategy = "mypackage.UUIDGenerator")
   public String getTransactionGuid()
...
```

In TopLink, a custom sequence generator can be implemented and registered by using the `@GeneratedValue` annotation. For more information, see: "How to use Custom Sequencing (i.e., UUID)" in the EclipseLink documentation.

http://wiki.eclipse.org/EclipseLink/Examples/JPA/CustomSequencing

10.2.3 Task 3: Convert Hibernate Mapping Annotations

The following sections describe how to convert various Hibernate annotations to TopLink annotations.

10.2.3.1 Convert the @ForeignKey Annotation

In Hibernate, the `@ForeignKey` annotation allows you to define the name of the foreign key to be used during schema generation.

TopLink does generate reasonable names, but does not provide an annotation or `eclipselink-orm.xml` support for specifying the name to use. When migrating, the recommended solution is to have TopLink generate the schema (DDL) commands to a script file instead of directly on the database. The script can then be customized to use different names prior to being executed.

> **Note:** The foreign key name is not used by TopLink at runtime, but is required if EclipseLink attempts to drop the schema. In this case, the drop script should be generated to a file and customized to match the foreign key names used during creation.

10.2.3.2 Convert the @Cache Annotation

In Hibernate, the `@Cache` annotation configures the caching of entities and relationships. Because TopLink uses an entity cache instead of a data cache, the relationships are automatically cached. In these cases, the `@Cache` annotation should simply be removed during migration.

When the `@Cache` annotation is used on an entity, its behavior is similar to TopLink's `@Cache` annotation. For more information on the `@Cache` annotation and equivalent `eclipselink-orm.xml` configuration values, see "Eclipse User Guide on JPA Extensions" in the EclipseLink documentation.

http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_Entity_Caching

10.2.4 Task 4: Modify the persistence.xml File

The `persistence.xml` file is the deployment descriptor file for JPA persistence. It specifies the persistence units, and declares the managed persistence classes, the Object-Relational mapping, and the database connection details. Example 10–3 illustrates a `persistence.xml` file for an application that uses Hibernate:

Example 10–3 Persistence File for an Application that uses Hibernate

```
<persistence>
   <persistence-unit name="helloworld">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
```

```
</persistence>
```

10.2.4.1 Modified persistence.xml

Example 10–4 illustrates a `persistence.xml` file modified for an application that uses TopLink. Key differences include the value for the persistence provider. For TopLink, this value is `org.eclipse.persistence.jpa.PersistenceProvider`. The names of TopLink-specific properties will typically be prefixed by `eclipselink`, for example, `eclipselink.target-database`.

Example 10–4 Persistence File Modified for EclipseLink

```
<xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="helloworld">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <!-- Entities must be specified for EclipseLink weaving -->
    <class>Todo</class>
    <properties>
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
      <property name="eclipselink.ddl-generation.output-mode" value="database"/>
      <property name="eclipselink.weaving" value="false"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

For more information, see "EclipseLink/Examples/JPA/Migration/JBoss" in the EclipseLink documentation.

http://wiki.eclipse.org/EclipseLink/Examples/JPA/Migration/JBoss

10.2.4.2 Drop and Create the Database

For production environments, you would normally have the schema setup on the database. The following properties defined in the persistence unit are more suitable for examples and demos. These properties will instruct TopLink to automatically drop and create database tables. Any previously existing tables will be removed.

```
<property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
<property name="eclipselink.ddl-generation.output-mode" value="database"/>
```

10.2.5 Task 5: Convert Hibernate API to EclipseLink API

Table 10–2 describes the Hibernate classes that are commonly used in a JPA project and their equivalent EclipseLink (JPA) interfaces. All of the Hibernate classes are in the `org.hibernate` package. All of the JPA interfaces (and the `Persistence` class) are in the `javax.persistence` package.

For information on the EclipseLink API, see *Oracle Fusion Middleware Java API Reference for Oracle TopLink*.

Table 10–2 Hibernate Classes and Equivalent JPA Interfaces

org.hibernate	javax.persistence	Description
`cfg.Configuration`	`Persistence`	A bootstrap class that configures the session factory (in Hibernate) or the entity manager factory (in JPA). It is generally used to create a single session (or entity manager) factory for the JVM.
`SessionFactory`	`EntityManagerFactory`	Provides APIs to open Hibernate sessions (or JPA entity managers) to process a user request. Generally, a session (or entity manager) is opened per thread processing client requests.
`Session`	`EntityManager`	Provides APIs to store and load entities to and from the database. It also provides APIs to get a transaction and create a query.
`Transaction`	`EntityTransaction`	Provides APIs to manage transactions.
`Query`	`Query`	Provides APIs to execute queries.

10.3 Additional Resources

For more information on migrating from Hibernate to EclipseLink, see "EclipseLink/Examples/JPA/Migration/Hibernate":

http://wiki.eclipse.org/EclipseLink/Examples/JPA/Migration/Hibernate

A

Installing Oracle TopLink

This appendix contains information about installing Oracle TopLink. It contains the following sections:

- Appendix A.1, "System Requirements and Certifications"
- Appendix A.2, "Installing a Stand Alone Instance of Oracle TopLink"
- Appendix A.3, "Installing Oracle TopLink and EclipseLink with Oracle WebLogic Server"
- Appendix A.4, "Installing Oracle TopLink with Oracle Containers for Java EE"
- Appendix A.5, "Installing EclipseLink with Oracle Containers for Java EE"

A.1 System Requirements and Certifications

The complete product requirements list and the latest certification information for 11g Release 1 (11.1.1.6.0) are available at:

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

A.1.1 Additional Requirements

TopLink requires a Java Virtual Machine (JVM) compatible with JDK 1.5.0 (or higher). TopLink also requires internet access to use URL-based schemas and hosted documentation.

A.2 Installing a Stand Alone Instance of Oracle TopLink

Follow these steps to install TopLink stand alone (including TopLink Foundation Library and TopLink Workbench). Before you proceed with the installation, back up all existing project data.

1. Unzip the TopLink Zip file (`toplink.zip`) into an empty directory. This is your new `<TOPLINK_HOME>` directory where Oracle TopLink 11g Release 1 will reside.

   When unzipped, additional steps are required to run the Oracle TopLink Workbench and other utilities. For more information, see "Configuring the TopLink Workbench Environment" in the Oracle Fusion Middleware Developer's Guide for Oracle TopLink.

2. When installation is complete, refer to *Oracle TopLink Release Notes*.

.

A.3 Installing Oracle TopLink and EclipseLink with Oracle WebLogic Server

The Oracle WebLogic Server installation includes both the `toplink.jar` and the `eclipselink.jar`. No additional installation is required. For configuration information, see "Integrating TopLink with Oracle WebLogic Server" in the Oracle Fusion Middleware Developer's Guide for Oracle TopLink.

A.4 Installing Oracle TopLink with Oracle Containers for Java EE

Follow these steps to install TopLink with OC4J (including TopLink Foundation Library and TopLink Workbench). Before you proceed with the installation, back up all existing project data.

1. Unzip the OC4J Zip file (`oc4j_extended.zip`) into an empty directory. This is your new `<ORACLE_HOME>` directory where OC4J 10g will reside.
2. Unzip the TopLink Zip file (`toplink.zip`) in your `<ORACLE_HOME>` directory where `<ORACLE_HOME>` is the directory where you installed OC4J 10g.

   When unzipped, additional steps are required to run the Oracle TopLink Workbench and other utilities. For more information, see "Configuring the TopLink Workbench Environment" in the Oracle Fusion Middleware Developer's Guide for Oracle TopLink.
3. When installation is complete, refer to *Oracle TopLink Release Notes*.

A.5 Installing EclipseLink with Oracle Containers for Java EE

EclipseLink (`eclipselink.jar`) is included with the TopLink installation, as described in "Installing Oracle TopLink with Oracle Containers for Java EE".

For information on manually installing EclipseLink with OC4J, refer to "Integrating EclipseLink with Oracle Containers for Java EE" in the EclipseLink User's Guide at `http://wiki.eclipse.org/Integrating_EclipseLink_with_an_Application_Server_(ELUG)#Integrating_EclipseLink_with_Oracle_Containers_for_J2EE_.28OC4J.29`