Table of Contents

EclipseLink Solutions Guide for EclipseLink

PDF | ePub

EclipseLink

Solutions Guide for EclipseLink 5.0.0

March 30, 2026 **Solutions Guide for EclipseLink**

Print date: December, 2022

Preface

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

Audience

A variety of engineers use EclipseLink. Users of EclipseLink are expected to be proficient in the use of technologies and services related to EclipseLink (for example, Jakarta Persistence API). This guide does not include details about related common tasks, but focuses on EclipseLink functionality.

Users of this guide include:

- Developers who want to develop applications using any of the following technologies for persistence services:
  - Jakarta Persistence API (JPA) 2.*n* plus EclipseLink JPA extensions
  - Java Architecture for XML Binding 2.*n* (JAXB) plus EclipseLink Object-XML extensions
  - EclipseLink Database Web Services (DBWS)

    Developers should be familiar with the concepts and programming practices of Java Platform, Standard Edition (Java SE platform), and Java Platform, Enterprise Edition (Jakarta EE platform).

    Developers using EclipseLink JPA should be familiar with the concepts and programming practices of JPA 2.2, as specified in the Java Persistence Architecture 2.2 specification at `http://jcp.org/en/jsr/detail?id=338`.

    Developers using EclipseLink 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 EclipseLink 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`.
- Administrators and deployers who want to deploy and manage applications using EclipseLink persistence technologies. These users should be familiar with basic operations of the chosen application server.

Related Documents

For more information, see the following documents:

- *EclipseLink Concepts*
- *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*
- *Developing Persistence Architectures Using EclipseLink Database Web Services Developer's Guide*
- *Developing JAXB Applications Using EclipseLink MOXy*
- *Java API Reference for EclipseLink*
- EclipseLink Documentation Center at `http://www.eclipse.org/eclipselink/documentation/`

Conventions

The following text conventions are used in this guide:

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

List of Examples

What's New in This Guide

The following topics introduce the new and changed features of EclipseLink and other significant changes that are described in this guide, and provides pointers to additional information.

New and Changed Features for Release 5.0.0

- Serialized object policy, for storing a serialized version of an entity into a single column in the database. See "Serialized Object Policy".
- Automated tuning, for a dynamic single tuning option. See "Automated Tuning" on page 18-8.

Other Significant Changes in this Document for Release 5.0.0

New and Changed Features for 12c (12.1.2)

EclipseLink 2.4.2 includes the following new and changed features that are documented in this book. This list does not necessarily include all new or changed features in this release. It only includes the new features that are documented in this book.

- Client isolation, where multiple application tenants may share database tables and schemas. This allows applications to manage entities for multiple tenants in the same application. See Chapter 14, "Tenant Isolation Using EclipseLink."
- JSON bindings, for converting objects directly to and from JavaScript Object Notation (JSON). This can be useful when creating RESTful services, using JSON messages with Java API for RESTful Web Services (JAX-RS) services. See Chapter 16, "Converting Objects to and from JSON Documents,"
- RESTful persistence, where Jakarta Persistence API (JPA) entities can be exposed through standards-based RESTful services such as JAX-RS, using either JSON or XML media. See Chapter 19, "Exposing JPA Entities Through RESTful Data Services."
- Support for TopLink Database Change Notification (DCN), which allows the database to notify TopLink of database changes so that cached objects can be invalidated in the shared cache. See Chapter 20, "Using Database Events to Invalidate the Cache."
- NoSQL database support, allowing objects to be mapped to non-relational (NoSQL) data sources. See Chapter 21, "Using EclipseLink with NoSQL Databases,".

For a complete list of the changes in this release, see `http://www.eclipse.org/eclipselink/releases/2.5.php`.

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

For this release of EclipseLink, this guide has been updated in several ways. Following are the

sections that have been added or changed.

- Moved installation information from appendix to Chapter 2, "Installing EclipseLink,".
- Added new chapter, Chapter 5, "Using EclipseLink with JBoss 7 Application Server," to describe how EclipseLink can be used with applications deployed to JBoss Application Server 7.1.
- Added new chapter, Chapter 6, "Using EclipseLink with IBM WebSphere Application Server," to describe how EclipseLink can be used with applications deployed to IBM WebSphere Application Server
- Added new chapter, Chapter 7, "Migrating from Native TopLink," to describe migrate applications using "native" TopLink object-relational mapping (ORM) APIs to the current EclipseLink APIs.
- Added information about data partitioning in Chapter 10, "Scaling Applications in Clusters."
- Split Chapter 11, "Providing Software as a Service." into four chapters:
  - Chapter 11, "Providing Software as a Service." This is now just an overview of the following three chapters.
  - Chapter 12, "Making JPA Entities and JAXB Beans Extensible"
  - Chapter 13, "Using an External MetaData Source"
  - Chapter 14, "Tenant Isolation Using EclipseLink" and also updated this chapter with information about Virtual Private Database (VPD) multi-tenancy and table-per-tenant multi-tenancy
- Added new chapter, Chapter 16, "Converting Objects to and from JSON Documents," to describe how to convert objects directly to and from JSON
- Added information about weaving to Chapter 17, "Testing JPA Outside a Container," to describe how to use the persistence unit JAR file to test an application outside the container (for instance, in applications for the Java Platform, Standard Edition (Java SE platform)).
- Added new chapter, Chapter 19, "Exposing JPA Entities Through RESTful Data Services," to describe how to expose JPA entities through Jakarta Persistence API-RESTful Services (JPA-RS), using either JSON or XML media.
- Added new chapter, Chapter 20, "Using Database Events to Invalidate the Cache," to describe how to use EclipseLink Database Change Notification (DCN) for shared caching in a JPA environment. DCN allows the database to notify EclipseLink of database changes. The changed objects are invalidated in the EclipseLink shared cache. Stale data can be discarded, even if other applications access the same data in the database.
- Added new chapter, Chapter 22, "Using EclipseLink with the Oracle Database," to describe how to use the Oracle Database features that are supported by EclipseLink.

Chapter 1. Introduction

EclipseLink is a powerful and flexible Java persistence framework for storing Java objects in a data store, such as a relational database or a NoSQL database, and for converting Java objects to XML or JSON documents. EclipseLink provides APIs and a run-time environment for implementing the persistence layer of Java Platform, Standard Edition (Java SE platform), and Java Platform, Enterprise Edition (Jakarta EE platform) applications.

EclipseLink implements Jakarta Persistence API (JPA), Java Architecture for XML Binding (JAXB), and other standards-based persistence technologies and also includes extensions beyond those standards. For more information about the EclipseLink project, see "Eclipse Persistence Services Project (EclipseLink) home" at http://www.eclipse.org/eclipselink/. For the EclipseLink Documentation, Center see http://www.eclipse.org/eclipselink/documentation/.

1.1. About This Guide

This guide, *Solutions Guide for EclipseLink*, documents a number of scenarios, or use cases, that illustrate EclipseLink features and typical EclipseLink 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 guide, you should already be familiar with the following:

- The concepts and programming practices of the Java SE platform and the Jakarta EE platform. In the current release, EclipseLink supports Jakarta EE 6. For more information, see the following.

  Java Java home page: http://www.oracle.com/us/technologies/java/index.html Jakarta EE 5 Tutorial: http://download.oracle.com/javaee/5/tutorial/doc/bnbpy.html **Jakarta EE 6 Tutorial:** http://download.oracle.com/javaee/6/tutorial/doc/bnbpy.html Any of the thousands of books and websites devoted to Java.

  + **Eclipse Integrated Development Environment**

    - Eclipse IDE: http://www.eclipse.org/

- EclipseLink from the Eclipse Foundation
  - EclipseLink project home: http://wiki.eclipse.org/EclipseLink
  - EclipseLink Documentation Center: https://www.eclipse.org/eclipselink/documentation/
- If you are working with EclipseLink JPA, you should be familiar with the concepts and programming practices of JPA 2.2, as specified in the *Jakarta Persistence API, Version 2.2* specification at http://jcp.org/en/jsr/detail?id=338.
- If you are working with EclipseLink JAXB, 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.

- If you are using JSON data-interchange format, you should be familiar with the concepts and programming practices of JSON, as described at http://www.json.org/. For XML, see http://www.w3.org/XML/
- If you are working with EclipseLink 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. If you are using the JavaScript Object Notation (JSON) data-interchange format, you should be familiar with the concepts and programming practices of JSON, as described at http://www.json.org/. For XML, see http://www.w3.org/XML/
- If you are working with EclipseLink DBWS, you 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.
- If you are working with REpresentational State Transfer (REST) service, you should be familiar with concepts and programming practices of REST, as specified in "JSR 311: JAX-RS: The Java API for RESTful Web Services" at http://jcp.org/en/jsr/detail?id=311.

1.3. The Use Cases

The use cases documented in this guide are as follows:

- Chapter 2, "Installing EclipseLink" - How to download and install EclipseLink.
- Chapter 3, "Using EclipseLink with WebLogic Server" - How to use EclipseLink with WebLogic Server.
- Chapter 4, "Using EclipseLink with GlassFish Server" - How to use EclipseLink with GlassFish Server.
- Chapter 5, "Using EclipseLink with JBoss 7 Application Server" - How to use EclipseLink with JBoss 7 Application Server.
- Chapter 6, "Using EclipseLink with IBM WebSphere Application Server" - How to use EclipseLink with IBM WebSphere Application Server.
- Chapter 7, "Migrating from Native TopLink" - How to how to migrate applications using native EclipseLink object-relational mapping (ORM) API to the current EclipseLink API.
- Migrating from Hibernate to EclipseLink - How to migrate applications from using Hibernate JPA to using EclipseLink JPA.
- Chapter 9, "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 10, "Scaling Applications in Clusters" - How to configure EclipseLink applications to ensure scalability in clustered application server environments.
- Chapter 11, "Providing Software as a Service"
  - Overview of EclipseLink Software as a Service (SaaS) features..
- Chapter 12, "Making JPA Entities and JAXB Beans Extensible" - How to make JPA entities or JAXB beans extensible.

- Chapter 13, "Using an External MetaData Source" - How to use an external metadata source.
- Chapter 14, "Tenant Isolation Using EclipseLink" - How to support multiple application tenants who share data sources, including tables and schemas.
- Chapter 15, "Mapping JPA to XML" - How to map JPA entities to XML using EclipseLink MOXy.
- Chapter 17, "Testing JPA Outside a Container" - How to test your EclipseLink JPA application outside the container.
- Chapter 18, "Enhancing Performance" - Getting the best performance out of EclipseLink.
- Chapter 19, "Exposing JPA Entities Through RESTful Data Services" - How to expose entities through RESTful services using EclipseLink Jakarta Persistence API for RESTful Services (JPA-RS).
- Chapter 20, "Using Database Events to Invalidate the Cache" - How to use EclipseLink Database Change Notification (DCN) for caching with a shared database in JPA.
- Chapter 21, "Using EclipseLink with NoSQL Databases" - How to use EclipseLink to map objects to non-relational (that is, no SQL) data sources.

Chapter 2. Installing EclipseLink

This chapter tells how to install EclipseLink.

EclipseLink is available in several distributions which are installed in a variety of ways, as described in the following sections:

- Prerequisites
- Installing EclipseLink for Java SE and Jakarta EE Development
- Installing EclipseLink NoSQL Support
- Installing EclipseLink with OSGi Support

2.1. Prerequisites

EclipseLink requires a Java Virtual Machine (JVM) compatible with JDK 1.6.0 (or higher). EclipseLink also requires internet access to use URL-based schemas and hosted documentation.

2.2. Installing EclipseLink for Java SE and Jakarta EE Development

Use the following procedures to install EclipseLink for Java SE and EE development. Before you proceed with the install, it is recommended that you back up any existing project data.

1. Set the following system environment variables before installing EclipseLink:
   - `JAVA_HOME` - `Set JAVA_HOME` to where you installed your Java SDK home directory. For example:
     - Windows example: `JAVA_HOME = C:\JDK`
     - UNIX example: `JAVA_HOME = ⋯:/usr/java/jdk`
   - `PATH` - Set `PATH` to include `JDK/bin` directory. For example:
     - Windows example: `PATH = C:\JDK\bin`
     - UNIX example: `PATH = ⋯:/usr/java/jdk/bin`
2. Download the EclipseLink install archive zip file, `eclipse-``ver_no`.zip`, from the EclipseLink downloads page at `http://www.eclipse.org/eclipselink/downloads/`
3. Unzip the downloaded file in the desired installation directory. When you unzip the file, you will find an `eclipselink` subdirectory, containing multiple subdirectories. This directory is your new `*ECLIPSELINK_HOME*` directory. For example:
   - Windows example: `*ECLIPSELINK_HOME*` = <INSTALL_DIR>/eclipselink`
   - UNIX example: `*ECLIPSELINK_HOME*` = ⋯:/usr/el/``INSTALL_DIR`/eclipselink`

2.3. Installing EclipseLink NoSQL Support

Support for NoSQL databases was introduced in EclipseLink 2.4.

To add support for NoSQL databases to EclipseLink, download and install `eclipselink-plugins-nosql-``ver_no`.zip` file from http://www.eclipse.org/eclipselink/downloads/. Use this bundle in conjunction with `eclipselink.jar` or the EclipseLink JPA bundles.

For information about NoSQL support, see Chapter 21, "Using EclipseLink with NoSQL Databases."

2.4. Installing EclipseLink with OSGi Support

EclipseLink JPA support in OSGi is provided by the Eclipse Gemini JPA project. For more information, including installation instructions, see http://wiki.eclipse.org/Gemini/JPA/Documentation.

Chapter 3. Using EclipseLink with WebLogic Server

This chapter describes how to use EclipseLink as the persistence provider for applications deployed to Oracle WebLogic Server.

The chapter includes the following sections:

Use Case

WebLogic Server developers, administrators, and user want to take advantage of all the persistence and transformation services provided by EclipseLink.

Solution

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

Components

- WebLogic Server 12*c* or later. WebLogic Server includes EclipseLink.

EclipseLink's core functionality is provided by EclipseLink, the open source persistence framework from the Eclipse Foundation. EclipseLink implements Jakarta Persistence API (JPA), Java Architecture for XML Binding (JAXB), and other standards-based persistence technologies, plus extensions to those standards. EclipseLink includes all of EclipseLink, plus additional functionality from Oracle.

- A compliant Java Database Connectivity (JDBC) database including Oracle Database, Oracle Express, MySQL, and so on.
- While it is not required, you may want to use a Jakarta EE integrated development environment (IDE) for convenience during development.

Samples

See the following EclipseLink samples 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`

3.1. Introduction to the Solution

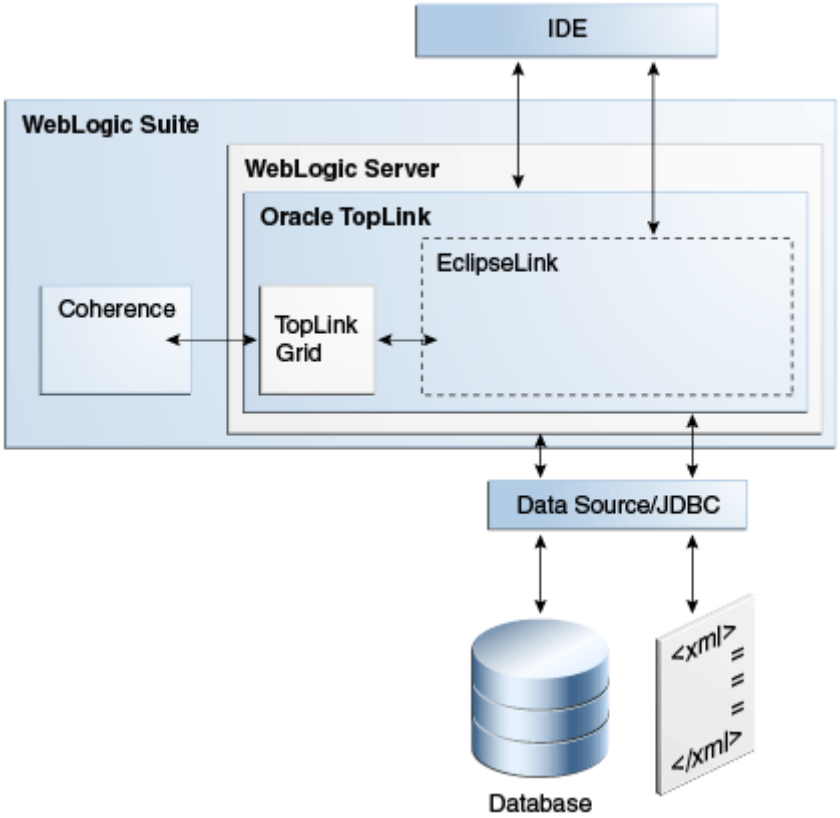
WebLogic Server is a scalable, enterprise-ready Java Platform, Enterprise Edition (Jakarta EE platform) application server. WebLogic Server's complete implementation of the Jakarta EE 6 specification provides a standard set of APIs for creating distributed Java applications that can access a wide variety of services, such as databases, messaging services, and connections to external enterprise systems. In addition to the Jakarta EE implementation, WebLogic Server enables enterprises to deploy 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*.

EclipseLink provides APIs and a run-time environment for implementing the persistence layer of Jakarta EE applications (as well as Java SE applications).

Advantages to Using EclipseLink with WebLogic Server

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

- EclipseLink is the default persistence provider for WebLogic Server domains, with support for JPA 2.1.
- The EclipseLink implementation of Java Architecture for XML Binding (JAXB) is the default JAXB implementation in WebLogic Server. EclipseLink fully implements JAXB and also includes other advanced features. By default, you can take advantage of EclipseLink JAXB in Java API for XML Web Services (JAX-WS) and Java API for RESTful Web Services (JAX-RS) applications.
- EclipseLink logging integration in WebLogic Server provides a comprehensive, integrated logging infrastructure. See Task 4: Use or Reconfigure the Logging Integration.



Description of "Figure 3-2 WebLogic Server, EclipseLink, Coherence and TopLink Grid integration"

3.2. Implementing the Solution

To run EclipseLink JPA applications in WebLogic Server, you must configure WebLogic Server and coordinate certain settings in it and in your application, as described in the following tasks:

- Task 1: Prerequisites
- Task 3: Configure JMX MBean Extensions in WebLogic Server
- Task 4: Use or Reconfigure the Logging Integration
- Task 5: Add Persistence to Your Java Application Using EclipseLink
- Task 6: Configure a Data Source
- Task 7: Extend the Domain to Use Advanced Oracle Database Features
- Task 8: Start WebLogic Server and Deploy the Application
- Task 9: Run the Application
- Task 10: Configure and Monitor Persistence Settings in WebLogic Server

Task 1: Prerequisites

This document 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.

- WebLogic Server 12*c* or later.

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

- Any compliant Java Database Connectivity (JDBC) database including Oracle Database, Oracle Express, MySQL, and so on.

  For Oracle Database, see http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html. For Oracle Database, 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, and NetBeans all provide sophisticated Jakarta EE development tools. Both JDeveloper and Oracle Enterprise Pack for Eclipse include embedded versions of WebLogic Server, although this guide describes a standalone instance of WebLogic Server.

  For JDeveloper, see http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html. For Oracle Enterprise Pack for Eclipse, see http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html. For NetBeans, see http://www.oracle.com/us/products/tools/050845.html.

Task 3: Configure JMX MBean Extensions in WebLogic Server

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

When deployed to WebLogic Server, EclipseLink applications deploy MBeans when they connect to the database, not at deployment time.

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

For information about EclipseLink logging in WebLogic Server, see Task 4: Use or Reconfigure the Logging Integration.

By default, when you deploy an EclipseLink application to WebLogic Server, the EclipseLink runtime deploys the following JMX MBeans to the WebLogic Server JMX service for each EclipseLink session:

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

Use the API that this JMX MBean exposes to access and configure your EclipseLink sessions at

runtime, using JMX code that you write, or to integrate your EclipseLink application with a third-party JMX management application, such as JConsole.

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

To enable anonymous lookup in the WebLogic Server 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 information about accessing the MBean information using various tools, see "Accessing Custom MBeans," in *Oracle Fusion Middleware Developing Manageable Applications With JMX for Oracle WebLogic Server*.

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

Task 4: Use or Reconfigure the Logging Integration

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

- *Oracle Fusion Middleware Using Logging Services for Application Logging for Oracle WebLogic Server*
- *Oracle Fusion Middleware Configuring Log Files and Filtering Log Messages for Oracle WebLogic Server*
- The logging topics in *Oracle Fusion Middleware 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`.

How the Logging Integration Works

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

As a result of this integration, EclipseLink logging levels are converted to WebLogic Server logging

levels as shown in Table 3-1.

Table 3-1 Mapping of EclipseLink Logging Levels to WebLogic Server Logging Levels

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

WebLogic Server logging levels are mapped to EclipseLink levels as shown in Table 3-2.

Table 3-2 Mapping of WebLogic Server Logging Levels to EclipseLink Logging Levels

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

Viewing Persistence Unit Logging Levels in the Administration Console

You can see the EclipseLink logging level defined for the persistence unit in the Administration Console, as described in Task 10: Configure and Monitor Persistence Settings in WebLogic Server. However, be aware that this logging level, set in the `persistence.xml` file, is overridden when the default WebLogic Server and EclipseLink logging integration is used. For information about overriding the integration, see Overriding the Default Logging Integration.

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

Overriding the Default Logging Integration

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

You can override the default logging integration by setting the `eclipselink.logging.logger` property name to a different setting. For example, to enable the default EclipseLink logging, set the `eclipselink.logging.logger` property as follows:

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

You can also use a different logging implementation for EclipseLink messages, for example the `java.util.logging` package:

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

Configuring WebLogic Server to Expose EclipseLink Logging

If you use the native EclipseLink logging implementation, you can still display EclipseLink 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 redirecting, see "Redirecting JVM Output" in *Oracle Fusion Middleware 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 Fusion Middleware Oracle WebLogic Server Administration Console Online Help*.

Other Considerations

Other things to consider include the following:

- The message ID `2005000` is used for all EclipseLink log messages.
- Some logging messages handled by the WebLogic Server 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 EclipseLink logging infrastructure and only the EclipseLink logging settings.
- If you use a different release of EclipseLink than the release bundled in your WebLogic Server installation (by using a filtering class loader), then trying to use 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.

Task 5: Add Persistence to Your Java Application Using EclipseLink

Using EclipseLink 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 guide. WebLogic Server imposes no special requirements on your EclipseLink application, other than the details described in this chapter.

This chapter describes features, settings, and tasks that are specific to using EclipseLink (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 `https://www.eclipse.org/eclipselink/documentation/`

- The *Jakarta Persistence API, Version 2.1* specification at `http://jcp.org/en/jsr/detail?id=317`
- "Part V, Persistence" in "The Jakarta EE 6 Tutorial" at `http://download.oracle.com/javaee/6/tutorial/doc/bnbpy.html`
- Any third-party book that describes programming Java applications using JPA

For more information about EclipseLink features and concepts, see Chapter 1, "Introduction" and *EclipseLink Concepts*.

For related WebLogic Server programming topics, see any book in the WebLogic Server documentation set, in particular the following:

- *Oracle Fusion Middleware Programming Enterprise JavaBeans, Version 3.0, for Oracle WebLogic Server*
- *Oracle Fusion Middleware Developing Applications for Oracle WebLogic Server*
- *Oracle Fusion Middleware Deploying Applications to Oracle WebLogic Server*
- *Oracle Fusion Middleware Programming JDBC for Oracle WebLogic Server*

Task 6: 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 Java Naming and Directory Interface (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 about using JDBC with WebLogic Server, see the following:

- For complete documentation about working with JDBC in WebLogic Server, see *Oracle Fusion Middleware 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 Server Administration Console, see the topics under "Configure JDBC" in *Oracle Fusion Middleware Oracle WebLogic Server Administration Console Online Help*.

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

Configure a Globally Scoped JTA Data Source

The most common data source configuration is a globally-scoped JNDI data source, using Java Transaction API (JTA) for transaction management, specified in the `persistence.xml` file. Configuration is straightforward, as shown in the following steps, and multiple applications can access the data source:

- Create the Data Source in WebLogic Server
- Configure the persistence.xml File

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 *Oracle Fusion Middleware Oracle WebLogic Server Administration Console Online Help*.

   EclipseLink 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 requirements. For more information, see "Create a JDBC Data Source" in *Oracle Fusion Middleware Oracle WebLogic Server Administration Console Online Help*.

   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 the `persistence.xml` file.

3. Configure connection pools, as described in "Configuring Connection Pool Features" in *Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*. The connection pool configuration can affect EclipseLink'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.

Configure the persistence.xml File

In the `persistence.xml` file, specify that `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 3-1:

Example 3-1 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>
```

Configure an Application-Scoped JTA Data Source

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

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

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 file. In that file, add `<scope>Application</scope>` to the `jdbc-data-source-params` section, as shown in Example 3-2.

***Example 3-2 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>
```

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 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 *Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*.

Add the JDBC Module to the WebLogic Server Application Configuration

Add a reference to the JDBC module in the `/META-INF/weblogic-application.xml` application deployment descriptor in the EAR file, as shown in Example 3-3. This registers the data source for use in the application.

Example 3-3 JDBC Module Defined in the weblogic-application.xml File

```
<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 *Oracle Fusion Middleware Deploying Applications to Oracle WebLogic Server* and "Enterprise Application Deployment Descriptor Elements" in *Oracle Fusion Middleware Developing Applications for Oracle WebLogic Server*.

Configure the JPA Persistence Unit to Use the JTA Data Source

To make it possible for EclipseLink 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 the `persistence.xml` file. For a JTA data source, add a fully qualified `jakarta.persistence.jtaDataSource` property, with the value `java:/app/jdbc/``*data_source_name*`, as shown in Example 3-4.

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

Example 3-4 JTA Data Source Definition in the persistence.xml File

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

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

To configure a non-JTA data source managed by the application, follow the procedures described in 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 3-5.

Example 3-5 non-JTA Data Source Definition in the persistence.xml File

```
<?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="jakarta.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 *Oracle Fusion Middleware Programming JTA for Oracle WebLogic Server*.

Ensure 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 the 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 you are using JTA, then the name must match `<jta-data-source>` in the `persistence.xml` file.
4. On the **Settings for *data_source_name*** > **Configuration** > **Connection Pool** page, review these settings:
   - The value for **URL** must match the `jakarta.persistence.jdbc.url` value in the `persistence.xml` file, for example, `jdbc:oracle:thin:@127.0.0.1:1521:XE`.
   - The value for **Driver Class Name** must match the `jakarta.persistence.jdbc.driver` value in the `persistence.xml` file, for example (for a JTA data source), `oracle.jdbc.xa.client.OracleXADataSource`.

Example 3-6 shows the values that must be shared in the domain's `config.xml` file and the application's `persistence.xml` file.

Example 3-6 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>
```

Task 7: Extend the Domain to Use Advanced Oracle Database Features

To fully support Oracle Spatial and Oracle XDB mapping capabilities (in both standalone WebLogic Server and the JDeveloper Integrated WebLogic Server), you must use the `toplink-spatial-template.jar` file and the `toplink-xdb-template.jar` file to extend the WebLogic Server domain to support Oracle Spatial and Oracle 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 3-3 and Table 3-4.

   Table 3-3 File to Support Oracle Spatial

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

   Table 3-4 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. Start 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 file (`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 *Oracle Fusion Middleware Domain Template Reference*.

Task 8: Start WebLogic Server and Deploy the Application

For information about deploying to WebLogic Server, see *Oracle Fusion Middleware Deploying Applications to Oracle WebLogic Server*. See also "Deploying Fusion Web Applications" in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Task 9: Run the Application

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

Task 10: 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 follows:

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 that 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 * 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 *Oracle Fusion Middleware Oracle WebLogic Server Administration Console Online Help*.

3.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- uniOracle WebLogic Server documentation
- *Java API Reference for EclipseLink*, including:
  - `org.eclipse.persistence`
  - `org.eclipse.persistence.jpa.PersistenceProvider`
  - `org.eclipse.persistence.services.mbean`

Chapter 4. Using EclipseLink with GlassFish Server

This chapter describes how to use EclipseLink as the persistence provider for applications deployed to Oracle GlassFish Server.

This chapter includes the following sections:

Use Case

Users want to run applications that employ JPA on Oracle GlassFish Server.

Solution

The Oracle GlassFish platform provides full support for EclipseLink. Developers writing applications for the GlassFish Server platform can achieve full Java-to-data source integration that complies with the Jakarta Persistence API (JPA) 2.0 specification. EclipseLink allows you to integrate Java applications with any data source, without compromising ideal application design or data integrity.

Components

- GlassFish Server 3.1.2.
- EclipseLink 2.3.0 or later.
- Any compliant JDBC database including Oracle Database, Oracle Database Express Edition, MySQL, and so on.
- While it is not required, you may want to use a Jakarta EE integrated development environment (IDE) for convenience during development.

4.1. Introduction to the Solution

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

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

Advantages to Using EclipseLink with GlassFish Server

By adding EclipseLink support, developers writing applications for the GlassFish Server platform can achieve full Java-to-data source integration that complies with the Jakarta Persistence API (JPA) 2.0 specification. EclipseLink allows you to integrate Java applications with any data source, without compromising ideal application design or data integrity. In addition, EclipseLink gives your GlassFish Server 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 EclipseLink can be used with other application servers, using GlassFish Server with EclipseLink provides a number of advantages:

- EclipseLink is included in all GlassFish Server distributions and is the default JPA provider.
- EclipseLink logging integration in GlassFish Server provides a comprehensive, integrated logging infrastructure.
- EclipseLink MOXy is also included in GlassFish versions 3.1.2 and later. Although it is not the default JAXB implementation, it can be used in JAX-WS and JAX-RS applications. For more information, see: `http://blog.bdoughan.com/2012/02/glassfish-312-is-full-of-moxy.html`

4.2. Implementing the Solution

To run EclipseLink 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: Prerequisites

This document 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.

- GlassFish Server 3.1.2.

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

- EclipseLink 2.4.1.

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

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

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

  For Oracle Database 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 Jakarta EE integrated development environment (IDE) for convenience during development. For example, Oracle JDeveloper, Oracle Enterprise Pack for Eclipse, and NetBeans all provide sophisticated Jakarta EE development tools.

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

  For Oracle Enterprise Pack for Eclipse, see http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html.

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

Task 2: Install GlassFish Server

EclipseLink is included with the GlassFish Server distribution. You can find instructions for installing and configuring GlassFish Server at this URL:

http://docs.oracle.com/cd/E26576_01/index.htm

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

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

- The `org.eclipse.persistence.oracle.jar` file is available with GlassFish and provides Oracle Database-specific functionality for EclipseLink. This file is used only for applications running against an Oracle Database.

Object-XML (also known as JAXB support, or *MOXy*) is a component that enables you to bind Java classes to XML schemas. This support is provided by the `org.eclipse.persistence.moxy.jar`.

Task 3: Set Up the Data Source

Configuring an Oracle database as a JDBC resource for a Jakarta EE application involves the following steps:

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

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 you do not have to restart GlassFish Server; the drivers are picked up dynamically.

   If the application uses Oracle Database-specific extensions provided by EclipseLink, then the driver must be copied to the `lib/ext` directory. For more information, see "Oracle Database Enhancements" in the *Oracle GlassFish Server Application Development Guide* at:

   `http://docs.oracle.com/cd/E26576_01/doc.312/e24930/jpa.htm#giqbi`

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

   To use the GlassFish Server Administration Console:

   In the GlassFish Server Administration Console, expand 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 Server Administration Console Online Help*.

   To start a standalone instance, expand the **Standalone Instances** node. For each instance that you are starting, select the instance in the Server Instances table. Click Start. The status of each instance is updated in the Server Instances table when the instance is started. For more information, see "To Start Standalone GlassFish Server Instances" in *GlassFish Server*

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
```

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, expand 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 Server 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=smith\\:Password=password\\:url=jdbc\\:oracle\\:thin\\:@node_name.example.com\
   \:1521\\:smithdb
       poolbvcallbackbmt
   ```

2. Verify connectivity to the database.

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

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, expand the **Resources** node, then the **JDBC** node, then the **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 Server Administration Console Online Help*.

To use the command line:

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

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

Task 4: Create the persistence.xml File

Example 4-1 illustrates a sample `persistence.xml` file that specifies the default persistence provider for EclipseLink, `org.eclipse.persistence.jpa.PersistenceProvider`. For more information about this file, see "About the Persistence Unit" in *EclipseLink Concepts*.

If you are using the default persistence provider, then you can specify additional database properties described in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

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 follows:

JDBC Data Source Properties:

- **Name**: The name of the data source, which is typically the same as the JNDI name, 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 Edition 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 Edition 11*g*.
- **Database User Name**: The database account user name used to create database connections, for example `hr` for Oracle Database Express Edition 11*g* samples.

- **Password**: Your password.

Select Targets:

- **Servers** / **Clusters**: Select the administration server, managed servers, or clusters to which you want to deploy the data source. You can choose one or more.

The sample `persistence.xml` file in Example 4-1 highlights the properties defining the persistence provider, the JTA data source, and logging details. In this example, the logging level is set to `FINE`. At this level, SQL code generated by EclipseLink is logged to the `server.log` file. For more information about these properties, see:

- Specify the Persistence Provider.
- Specify an Oracle Database.
- Specify Logging.

Example 4-1 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>
```

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 EclipseLink is `org.eclipse.persistence.jpa.PersistenceProvider`.

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 nondefault 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`, then 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.

For more information about specifying database properties in a `persistence.xml` file for GlassFish Server, see "Specifying the Database for an Application" in the *Oracle GlassFish Server Application Development Guide*, at:

http://docs.oracle.com/cd/E26576_01/doc.312/e24930/jpa.htm#gbwmj

Specify Logging

EclipseLink provides a logging utility even though logging is not part of the JPA specification. Hence, the information provided by the log is EclipseLink JPA-specific. With EclipseLink, 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. EclipseLink 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 level to use.

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

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

EclipseLink'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, EclipseLink is configured by GlassFish Server to use `JavaLogger` by default. The log is always redirected to the GlassFish Server `server.log` file. For more information, see "Setting Log Levels" in *Oracle GlassFish Server Administration Guide*, at:

http://docs.oracle.com/cd/E26576_01/doc.312/e24928/logging.htm#gklml

For more information about EclipseLink logging and the levels of logging available in the logging utility, see "Persistence Property Extensions Reference" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Task 5: Set Up GlassFish Server for JPA

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

http://docs.oracle.com/cd/E26576_01/doc.312/e24930/jpa.htm

It provides more information about these topics:

- "Specifying the Database for an Application," for information about database connection properties
- "Specifying the Persistence Provider for an Application," for setting the default or non-default persistence provider for an application
- "Primary Key Generation Defaults," for the default persistence provider's primary key generation defaults
- "Automatic Schema Generation," for information on annotations and options to manage automatic schema generation
- "Restrictions and Optimizations," for restrictions and performance optimizations that affect using the Jakarta Persistence API

Task 6: Create the Application

To create an application that uses EclipseLink as its JPA persistence provider, you may want to use a Jakarta EE IDE for convenience during development. For example, JDeveloper, Oracle Enterprise Pack for Eclipse, and NetBeans provide sophisticated Jakarta EE development tools, including support for EclipseLink. See "Key Tools" in *EclipseLink Concepts*.

For guidance in writing your application, see these topics from the "Configuring the Java Persistence Provider" chapter in *Oracle GlassFish Server Application Development Guide*, at:

http://docs.oracle.com/cd/E26576_01/doc.312/e24930/jpa.htm

Task 7: 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 *GlassFish Server Administration Console Online Help*. See also *Oracle GlassFish Server Application Deployment Guide*, at:

http://docs.oracle.com/cd/E26576_01/index.htm

Task 8: Run the Application

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

Task 9: Monitor the Application

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

`http://docs.oracle.com/cd/E26576_01/doc.312/e24928/monitoring.htm`

4.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- Oracle GlassFish Server Administration Guide

  `http://docs.oracle.com/cd/E26576_01/doc.312/e24928/toc.htm`

- Oracle GlassFish Server Application Deployment Guide

  `http://docs.oracle.com/cd/E26576_01/doc.312/e24929/toc.htm`

- Oracle GlassFish Server Application Development Guide

  `http://docs.oracle.com/cd/E26576_01/doc.312/e24930/toc.htm`

- Oracle GlassFish Server 3.1.2 to 3.1.2.2 Documentation Library

  `http://docs.oracle.com/cd/E26576_01/index.htm`

Chapter 5. Using EclipseLink with JBoss 7 Application Server

This chapter introduces and describes how to use EclipseLink as the persistence provider for applications deployed to JBoss Application Server 7.1.

This chapter includess the following sections:

Use Case

EclipseLink can be used with a number of popular Jakarta EE application servers, including JBoss Application Server.

Solution

Configure JBoss to use EclipseLink runtime, and deploy applications developed using EclipseLink APIs.

Components

- EclipseLink 2.4 or later.
- JBoss Application Server 7.x.
- A compliant Java Database Connectivity (JDBC) database, such as Oracle Database, Oracle Express, MySQL, the HSQL database embedded in JBoss Application Server, etc.

5.1. Introduction to the Solution

JBoss Application Server implements the Java Platform, Enterprise Edition (Jakarta EE). JBoss 7 fully supports Jakarta EE 6, while JBoss 6 officially supports only the Jakarta EE 6 Web Profile.

By configuring JBoss to support EclipseLink, you can take advantage of EclipseLink's full support for Jakarta Persistence API (JPA), Java Architecture for XML Binding (JAXB), including EclipseLink's extensions to those technologies, as well as EclipseLink Database Web Services (DBWS) to access to relational database artifacts via a Web service.

5.2. Implementing the Solution

To develop, deploy and run EclipseLink applications in JBoss Application Server 7, you must create EclipseLink as a module of JBoss. You must also create other modules, such as a JDBC driver, etc., in order to run applications.

This section contains the following tasks for using EclipseLink with JBoss 7.1:

Task 1: Prerequisites

Ensure that you have installed the following components:

- JBoss, version 7 or later. These instructions are based on JBoss release 7.1.1.

  Download JBoss from `http://www.jboss.org/jbossas/downloads/` . The version of JBoss must be identified as "Certified Jakarta EE6." Version 7.1.1 or later is recommended.

- EclipseLink 2.4 or later.

  Download EclipseLink from `http://www.eclipse.org/eclipselink/downloads/`.

- Any compliant Java Database Connectivity (JDBC) database including Oracle Database, Oracle Express, MySQL, the HSQL database embedded in JBoss Application Server, and so on.

Oracle XML DB (XDB) and JBoss Application Server both use port `8080` by default. If you have both available at the same URI, for example `localhost`, you must reconfigure one or the other to use a different port, for example `8081`.

For the Oracle Database, see `http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html`. For the Oracle Database, 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`. For information about the embedded HSQL database, see the JBoss documentation. * 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, and NetBeans all provide sophisticated Jakarta EE development tools that support EclipseLink.

For JDeveloper, see `http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html`. For Oracle Enterprise Pack for Eclipse, see `http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html`. For NetBeans, see `http://www.oracle.com/us/products/tools/050845.html`.

Task 2: Configure EclipseLink as a Module in JBoss

To configure EclipseLink as a module in JBoss:

1. Create a directory as follows:

   `*JBOSS_HOME*``\modules\org\eclipse\persistence\main`

2. Copy `eclipselink.jar` to the directory created in step 1. (The `eclipselink.jar` file is located in the `eclipselink/jlib` directory of the `eclipselink-``ver_no`.zip` file.)

3. Create a `module.xml` file in the directory created in step 1, with the following content:

```
<module xmlns="urn:jboss:module:1.1" name="org.eclipse.persistence">
   <resources>
       <resource-root path="eclipselink.jar"/>
       <!-- Insert resources here -->
   </resources>
   <dependencies>
      <module name="javax.api"/>
      <module name="jakarta.persistence.api"/>
      <module name="javax.transaction.api"/>
      <module name="javax.validation.api"/>
      <module name="jakarta.xml.bind.api"/>
      <module name="org.antlr"/>
      <module name="org.apache.commons.collections"/>
      <module name="org.dom4j"/>
      <module name="org.javassist"/>
      <module name="org.jboss.logging"/>
      <module name="com.oracle.ojdbc6"/>
   </dependencies>
</module>
```

Task 3: Add ojdbc6.jar as a Module in JBoss

Add the Oracle thin driver `ojdbc6.jar` as a module within JBoss, as follows:

1. Create the module directory:

   `*JBOSS_HOME*``\modules\com\oracle\ojdbc6\main`

2. Copy `ojdbc6.jar` to the module directory created in step 1.

3. Create a `module.xml` file in the module directory created in step 1, with the following contents:

```
<module xmlns="urn:jboss:module:1.1" name="com.oracle.ojdbc6">
    <resources>
        <resource-root path="ojdbc6.jar"/>
        <!-- Insert resources here -->
    </resources>
    <dependencies>
        <module name="javax.api"/>
    </dependencies>
</module>
```

Task 4: Create the Driver Definition and the Datasource

Create the driver definition and create the datasource.

The following instructions tell how to configure JBoss for running in standalome mode, using the `standalone.xml` configuration file. For instructions on how to use `domain.xml` to configure JBoss for running in domain mode, see the JBoss documentation.

1. In the standalone configuration file `JBOSS_ HOME`\standalone\configuration\standalone.xml`, find the following:

   ```
   <subsystem xmlns="urn:jboss:domain:datasources:1.0">
   ```

2. In that section, configure the datasource. The following example shows a configuration for the Oracle Database, using the Oracle JDBC Thin driver. For instructions on configuring other datasources, see the JBoss documentation.

```
<subsystem xmlns="urn:jboss:domain:datasources:1.0">
   <datasources>
      <datasource jndi-name="java:/EclipseLinkDS"
            pool-name="EclipseLinkDS"
            enabled="true"
            jta="true"
            use-java-context="true"
            use-ccm="true">
         <connection-
url>jdbc:oracle:thin:node_name.example.com:1521:TOPLINK</connection-url>
         <driver>oracle</driver>
         <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-isolation>
         <pool>
            <prefill>true</prefill>
            <use-strict-min>false</use-strict-min>
            <flush-strategy>FailingConnectionOnly</flush-strategy>
         </pool>
         <security>
            <user-name>Smith</user-name>
            <password>password</password>
         </security>
       </datasource>
       <driver name="oracle" module="com.oracle.ojdbc6">
         <xa-datasource-class>oracle.jdbc.OracleDriver</xa-datasource-class>
       </driver>
    </datasources>
</subsystem>
```

Task 5: Create Users

Starting with JBoss Application Server 7.1, you must create an Application User to get started, because remote access to the JNDI tree is secured by default, and you must provide login credentials. Therefore, at a minimum, you just create an Application User to be able to deploy an

application to the server. If you want to use the JBoss administration console for administration tasks, for example to view the JNDI tree, you must also create an Administration User.

To create user credentials, use the JBoss `add-user.bat` utility, located in `*JBOSS_HOME*``\bin\`.

For more information about security in JBoss Application Server, refer to the JBoss documentation.

Task 6: Modify JBoss Properties

Modify JBoss properties, as follows:

```
## JBoss-7.x
server.factory=org.jboss.naming.remote.client.InitialContextFactory
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
server.depend=jboss-client.jar
jboss.server=${jboss.home}/standalone
server.lib=${jboss.home}/bin/client
server.url=remote://localhost:4447
server.user=usera
server.pwd=passworda
jboss.naming.client.ejb.context=true
```

Task 7: Other Requirements

1. Add `junit.jar` in the `ear` under the `\lib` directory.
2. Because of a classloading issue in JBoss, you must list all your entity classes in `persistence.xml`. You can use either `<class>` elements or a global `<exclude-unlisted-classes>false</exclude-unlisted-classes>` element.
3. Add both `jndi.properties` and `jboss-ejb-client.properties` in the client classpath.

Task 8: Start JBoss

Start JBoss by running `standalone.bat` (for a single-server configuration) or `domain.bat` file (in a clustered environment) in `*JBOSS_ HOME*``\bin\`.

For information on different ways to configure and start JBoss, see the JBoss documentation.

5.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- JBoss Community at `http://www.jboss.org`.

Chapter 6. Using EclipseLink with IBM WebSphere Application Server

This chapter describes how to use EclipseLink as the persistence provider for applications deployed to IBM WebSphere Application Server.

This chapter includes the following sections:

Use Case

EclipseLink can be used with a number of popular Jakarta EE application servers, including WebSphere Application Server.

Solution

Configure WebSphere to use EclipseLink runtime, and deploy applications developed using EclipseLink APIs.

Components

- EclipseLink 2.4 or later.
- WebSphere Application Server 7 or later. These instructions are based on WebSphere 8.5.
- A compliant Java Database Connectivity (JDBC) database, such as Oracle Database, Oracle Express, MySQL, the Derby database included in WebSphere Application Server, and so on.

6.1. Introduction to the Solution

WebSphere Application Server implements Java Platform, Enterprise Edition (Jakarta EE). WebSphere V8.5 fully supports Jakarta EE 6 and can support Java Platform, Standard Edition (Java SE) 7 via a plugin.

By configuring WebSphere support EclipseLink, you can create and deploy applications that take advantage of EclipseLink's full support for Jakarta Persistence API (JPA), as well as EclipseLink's many extensions.

6.2. Implementing the Solution

To develop, deploy, and run EclipseLink applications in IBM WebSphere, you must add various modules including EclipseLink to WebSphere, and you must configure various aspects of WebSphere to support EclipseLink.

This section contains the following tasks for using EclipseLink with IBM WebSphere, Version 7 or

later:

- Task 1: Prerequisites
- Task 2: Configure Persistence Units
- Task 3: Configure the Server and the Application to Use EclipseLink

Task 1: Prerequisites

Ensure that you have installed the following components:

- IBM WebSphere, Version 7 or later. These instructions are based on WebSphere, Version 8.5.

  Obtain IBM WebSphere from `http://www-01.ibm.com/software/webservers/appserv/was/`.

- EclipseLink 2.4 or later.

  Download EclipseLink from `http://www.eclipse.org/eclipselink/downloads/`.

  You will use the following files: `eclipselink.jar jakarta.persistence_``ver_no`.jar`

Task 2: Configure Persistence Units

Configure persistence units to use EclipseLink as the persistence provider and to use WebSphere as the target server.

Example 6-1 shows a sample configuration for a container-managed persistence unit.

Example 6-1 Sample persistence.xml for a container-managed 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="default" transaction-type="JTA">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <jta-data-source>jdbc/EclipseLinkDS</jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.target-server" value="WebSphere_7"/>
            <property name="eclipselink.target-database"

value="org.eclipse.persistence.platform.database.oracle.Oracle11Platform"/>
            <property name="eclipselink.validate-existence" value="true"/>
            <property name="eclipselink.weaving" value="true"/>
            <property name="eclipselink.logging.level" value="FINEST"/>
        </properties>
    </persistence-unit>
</persistence>
```

Example 6-2 shows a sample configuration for an application-managed persistence unit.

Example 6-2 Sample persistence.xml for an application-managed 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="default" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <non-jta-data-source>jdbc/ELNonJTADS</non-jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.target-server" value="WebSphere_7"/>
            <property name="eclipselink.target-database"

value="org.eclipse.persistence.platform.database.oracle.Oracle11Platform"/>
            <property name="eclipselink.validate-existence" value="true"/>
            <property name="eclipselink.weaving" value="true"/>
            <property name="eclipselink.logging.level" value="FINEST"/>
        </properties>
    </persistence-unit>
</persistence>
```

Note the following about the two examples above:

- The `eclipselink.target-server` value `WebSphere_7` is used for WebSphere Application Server version 7 and later.
- Specifying `persistence_1_0.xsd" version="1.0"` for the persistence schema version works with both JPA 1 and JPA 2. For a JPA 2.*n* -only application, you can change the version to `persistence_2_0.xsd" version="2.`n`"` (WebSphere's support for JPA 2 began in WebSphere Application Server 7.0.0.9.

Task 3: Configure the Server and the Application to Use EclipseLink

The following are typical scenarios for using EclipseLink with the application server:

- **Modify Server to Make EclipseLink Available Globally**
- **Package EclipseLink in the Application EAR**
- **Package EclipseLink in the WAR**

Modify Server to Make EclipseLink Available Globally

You can make EclipseLink available globally for both container-managed and application-managed persistence units in either of the following ways:

- Option 1: Create a Global Shared Library (Recommended)
- Option 2: Add EclipseLink as a Server Library Extension

Option 1: Create a Global Shared Library (Recommended)

1. Create a global shared library containing the following files:
   - `eclipselink.jar`

     Find this file in the `*TOPLINK_INSTALLATION*` `\oracle_common\modules\oracle.toplink_*ver_no*` directory created by the EclipseLink quick installer.

   - `xmlparserv2.jar`

     Find this file in the `*TOPLINK_INSTALLATION*``\toplink\modules` directory created by the s quick installer.

   - If you use Oracle Database features such as `NCHAR`, `XMLTYPE`, and `MDSYS.SDO_GEOMETRY` with JPA, you must also include `xdb.jar` and `sdoapi.jar` in the shared library. Those files are available in your Oracle Database distribution.

     See the WebSphere documentation for instructions on how to use WebSphere to facilitate the creation of shared libraries.

2. Associate the shared library with the application.

   See the WebSphere documentation for instructions on how to use WebSphere to associate the shared library with an application.

Option 2: Add EclipseLink as a Server Library Extension

To add EclipseLink as a server library extension, copy `eclipselink.jar` and the other JAR file(s) listed in Option 1, above, to the `*WAS_HOME*``\lib\ext` directory.

Package EclipseLink in the Application EAR

You can also implement container-managed persistence by adding `eclipselink.jar` in the application EAR, without making any modifications to the server configuration. In this case, the persistence unit is managed by `@PersistenceContext` entity manager proxy injection on a stateless session bean. The following instructions show a example of this approach.

1. Add `eclipselink.jar` to the application EAR in the following location:

   `*EAR_archive*``/APP-INF/lib/`

2. Add the path to the `eclipselink.jar` to the `*ejbModule*``/META-INF/MANIFEST.MF` file(s) in your EJB JAR(s), as shown below:

   ```
   Manifest-Version: 1.0
   Class-Path: APP-INF/lib/eclipselink.jar
   ```

   This is the manifest at the root of the entities' location, in this case as part of the `ejb.jar.`

3. Configure the class loader to load the classes with the application class loader first.
4. Deploy and start the application. See the IBM WebSphere documentation for instructions.

Package EclipseLink in the WAR

If you do not or cannot implement container-managed persistence, as described in the previous two scenarios, you can create an application managed entity manager. In this case, all library configuration and classloader scope changes must be done inside the EAR itself.

1. Add `eclipselink.jar` and `jakarta.persistence_``ver_no`.jar` to the web application archive (WAR) file in the following location:

   `*WAR_archive*``/WEB-INF/lib/`

2. Configure the class loader order for your application to load the classes with the application class loader first. See the WebSphere documentation for instructions on setting class loader order using the Administrative console.
3. Deploy and start the application. See the IBM WebSphere documentation for instructions.

6.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- WebSphere Application Server at http://www-01.ibm.com/software/webservers/appserv/was/.

Chapter 7. Migrating from Native TopLink

This chapter describes how to migrate applications using "native" TopLink object-relational mapping (ORM) APIs to the current EclipseLink APIs.

This chapter includes the following sections:

Use Case

A developer wants to upgrade an application that uses the older TopLink native ORM to use a current EclipseLink ORM implementation.

Solution

Follow the instructions in this chapter to upgrade the application.

Components

- EclipseLink 2.4 or later.
- (Optional) EclipseLink Workbench.

7.1. Introduction to the Solution

"Native" TopLink ORM refers to the API, configuration files, and tools for object-relational mapping that evolved in TopLink before the Jakarta Persistence API (JPA) standardized an object-relational mapping API. Full JPA support was introduced in Oracle TopLink 10*g* (10.1.3.1.0), via TopLink Essentials. However, native TopLink continued to be supported.

Prior to the TopLink 11*g* (11.1.1) release, Oracle contributed the TopLink source code—including TopLink JPA and native TopLink—to the Eclipse Foundation, where it was used to form the basis of the open-source EclipseLink persistence services project. Then, in TopLink 11*g* Release 1 (11.1.1), Oracle started to include EclipseLink in TopLink, providing TopLink's core functionality.

EclipseLink developers using TopLink versions 11.1.1.0.0 though 11.1.1.6.0 have access to native TopLink ORM in either the proprietary Oracle `toplink.jar` or in the EclipseLink `eclipselink.jar`. In `toplink.jar`, the classes are in packages whose names start with `oracle.toplink.*`. In `eclipselink.jar`, those package names begin instead with `org.eclipselink.persistence.`.

The `toplink.jar` file was deprecated in TopLink 11*g* and is no long shipped with TopLink 12*c*. It is recommended that you migrate off `oracle.toplink.*` in TopLink 11*g*.

You can migrate applications that use `oracle.toplink.*` packages from `toplink.jar` to use `org.eclipselink.persistence.` packages from `eclipselink.jar`. The application functionality remains

the same, but migrating to `eclipselink.jar` provides the most up-to-date code base. After migrating, you will have access to other EclipseLink features and will be better prepared to convert your application to use JPA or one of the other persistence services included in current versions of EclipseLink.

This chapter explains how to use the renaming tool that is packaged with stand-alone EclipseLink to easily change the package names in your application and how to perform other actions necessary to migrate to the current code base.

Following the instructions in this chapter will update your application to use the current EclipseLink code base. Doing so retains the design and functionality of your application as originally implemented. However, these instructions do not describe how to convert a native TopLink-based application to use JPA or any of the other persistence services in current versions of EclipseLink. See the other EclipseLink documentation sources for that information.

7.2. Implementing the Solution

This section contains the following tasks:

- Task 1: Prerequisites
- Task 2: Replace Deprecated and Removed Native APIs
- Task 3: Rename Packages
- Task 4: Convert XML Configuration Files

Task 1: Prerequisites

- EclipseLink 2.4 or later.

  Download EclipseLink from `http://www.eclipse.org/eclipselink/downloads/`.

- (Optional) EclipseLink Workbench. The EclipseLink Workbench is available in EclipseLink downloads. See the EclipseLink download page at `http://www.eclipse.org/eclipselink/downloads/`.

Task 2: Replace Deprecated and Removed Native APIs

APIs that were deprecated in releases before TopLink 11g Release 1 (11.1.1) were removed in EclipseLink. If your application uses any of those deprecated APIs or any APIs that were already replaced or removed from TopLink, you must update the application to use current APIs.

The following sections lists the replaced and removed APIs, with suggested substitutions:

- APIs Replaced,
- Deprecated APIs,
- Removed API,

When suggested replacements are in `oracle.toplink.*` packages, you must also change the package names, as described in Task 3: Rename Packages.

APIs Replaced

The following tables list the APIs removed as of TopLink 11*g* Release 1 (11.1.1.1.). Use the replacement API listed in the tables.

Table 7-1 changetracking (oracle.toplink.descriptors.)*

Class Name	Method Name	Replacement APIs
`ChangeTracker`	`getTopLinkPropertyChangeListener`	`._persistence_getPropertyChangeListener`
`ChangeTracker`	`setTopLinkPropertyChangeListener(PropertyChangeListener)`	`_persistence_setPropertyChangeListener(PropertyChangeListener)`

Table 7-2 databaseaccess (oracle.toplink.internal)*

Class Name	Method Name	Replacement APIs
`.*Platform`	Whole class	`oracle.toplink.platform.database.*Platform`

Table 7-3 jdo (oracle.toplink.)*

Class Name	Method Name	Replacement APIs
.jdo	Whole package	None

Table 7-4 mappings (oracle.toplink.)*

Class Name	Method Name	Replacement APIs
TypeConversionMapping	Whole class	.mappings.converters.TypeConversionConverter
ObjectTypeMapping	Whole class	.mappings.converters.ObjectTypeConverter
SerializedObjectMapping	Whole class	.mappings.converters.SerializedObjectConverter

Table 7-5 objectrelational (oracle.toplink.)*

Class Name	Method Name	Replacement APIs
Oracle8Platform	Whole class	oracle.toplink.platform.database.oracle.Oracle8Platform

Table 7-6 oraclespecific (oracle.toplink.)*

Class Name	Method Name	Replacement APIs
.oraclespecific.NCharacter	Whole class	.platform.database.oracle.NCharacter
.oraclespecific.NClob	Whole class	.platform.database.oracle.NClob
.oraclespecific.NClob	Whole class	.platform.database.oracle.NClob
.oraclespecific.Oracle8Platform	Whole class	.platform.database.oracle.Oracle8Platform
.oraclespecific.Oracle9Specific [Foot 1]	Whole class	.platform.database.oracle.Oracle9Specific
.oraclespecific.TopLinkXMLType [Foot 2]	Whole class	None

[Footnote 1] `oracle.toplink.oraclespecific.Oracle9Specific` was moved to an internal package and renamed to `oracle.toplink.internal.platform.database.oracle.Oracle9Specific`. The replacement public API for `oracle.toplink.oraclespecific.Oracle9Specific` is `oracle.toplink.platform.database.oracle.Oracle9Specific`.

[Footnote 2] `oracle.toplink.oraclespecific.TopLinkXMLType` was a miscellaneous class, which does not have a replacement API.

Table 7-7 publicinterface (oracle.toplink.)*

Class Name	Method Name	Replacement APIs
DatabaseRow	Whole class	.sessions.DatabaseRecord

`DatabaseSession` [Foot 1]	Whole class	`.sessions.DatabaseSession`
`Descriptor`	Whole class	`.descriptors - ClassDescriptor, RelationalDescriptor`
`DescriptorEvent`	Whole class	`.descriptors.DescriptorEvent`
`DescriptorEventListener`	Whole class	`.descriptors` - new interface will not extend old interface
`DescriptorEventManager`	Whole class	`.descriptors`
`DescriptorQueryManager`	Whole class	`.descriptors`
`InheritancePolicy`	Whole class	`.descriptors`
`Session` [Foot 2]	Whole class	`.sessions.Session`
`UnitOfWork` [Foot 3]	Whole class	`.sessions.UnitOfWork`

Footnote 1 `oracle.toplink.publicinterface.DatabaseSession` was moved to an internal package and renamed to `oracle.toplink.internal.sessions.DatabaseSessionImpl`. The replacement public API for `oracle.toplink.publicinterface.DatabaseSession` is `oracle.toplink.sessions.DatabaseSession`.
Footnote 2 `oracle.toplink.publicinterface.Session` was moved to an internal package and renamed to `oracle.toplink.internal.sessions.AbstractSessionImpl`. The replacement public API for `oracle.toplink.publicinterface.Session` is `oracle.toplink.sessions.Session`.
Footnote 3 `oracle.toplink.publicinterface.UnitOfWork` was moved to an internal package and renamed to `oracle.toplink.internal.sessionl.UnitOfWorkImpl`. The replacement public API for `oracle.toplink.publicinterface.UnitOfWork` is `oracle.toplink.sessions.UnitOfWork`.

Table 7-8 sdk (oracle.toplink.)*

Class Name	Method Name	Replacement APIs
`.sdk`	Whole package	`.eis`

Table 7-9 entitymanager (oracle.toplink.sessions.)*

Class Name	Method Name	Replacement APIs
All classes	All methods	JPA: see JPA Persistence Provider Implementation,

Table 7-10 sessionconfiguration (oracle.toplink.tools.)*

Class Name	Method Name	Replacement APIs
`WASXMLLoader`	All methods	None

Table 7-11 xml (oracle.toplink.)*

Class Name	Method Name	Replacement APIs
`.xml`	Whole package	`.ox`

.xmlstream	Whole package	.ox
.xml.tools	Whole package	.ox
.xml.xerces	Whole package	.ox
.xml.zip	Whole package	.ox

*Table 7-12 XMLCommandConverter (oracle.toplink.)**

Class Name	Method Name	Replacement APIs
.remotecommand.XMLCommandConverter	Whole class	None
.transform.xml.XMLSource	Whole class	None
.transform.xml.XMLResult	Whole class	None
.internal.localization.i18n.ExceptionLocalizationResource	"error_loading_resources"	None
.internal.localization.i18n.ExceptionLocalizationResource	"error_parsing_resources"	None
.internal.localization.i18n.ExceptionLocalizationResource	"unexpect_argument"	None

*Table 7-13 Remote Protocols (oracle.toplink.)**

Class Name	Method Name	Replacement APIs
.remote.corba.orbix	Whole package	None
.remote.corba.visibroker	Whole package	None
.remote.ejb	Whole package	None
.tools.sessionconfiguration.TopLinkSessionsFactory	References for any of JNDIClusteringService in orbix, visibroker and ejb packages.	None
.tools.sessionconfiguration.DTD2SessionConfigLoader	References for any of JNDIClusteringService in orbix, visibroker and ejb packages.	None
.tools.sessionconfiguration.model.clustering.VisibrokerCORBAJNDIClusteringConfig	Whole class	None
.tools.sessionconfiguration.model.clustering.OrbixCORBAJNDIClusteringConfig	Whole class	None
.tools.sessionconfiguration.model.clustering.EJBJNDIClusteringConfig	Whole class	None
.tools.sessionconfiguration.XMLSessionConfigProject	References for any of JNDIClusteringService in orbix, visibroker and ejb packages.	None

Table 7-14 EJB Mapping for BEA WebLogic 6.1

Class Name	Method Name	Replacement APIs
`toplink-cmp-bean_name.xml`	None	A warning will be added at the beginning of: `internal.ejb.cmp.wls11.CMPDeployer.readTypeSpecificData()`

Deprecated APIs

The following tables list the APIs deprecated in the releases prior to TopLink 11*g* Release 1 (11.1.1) and therefore removed in that release, due to the substitution of EclipseLink libraries. Use the replacement API indicated.

Because deprecated classes and moved classes have the same name, you may get compile errors if you use `import *` to import classes from both the old package and the new package. To avoid these errors, use `import` with a fully qualified package name.

- Table 7-15, "mappings (oracle.toplink.*)"
- Table 7-16, "descriptors (oracle.toplink.*)"

*Table 7-15 mappings (oracle.toplink.)**

Class Name	Method Name	Replacement APIs
`OneToOneMapping`	`useJoining`	`ForeignReferenceMapping.setJoinFetch(int)`

*Table 7-16 descriptors (oracle.toplink.)**

Class Name	Method Name	Replacement APIs
`ClassDescriptor`	`addMultipleTableForeignKeyField`	`addForeignKeyFieldForMultipleTable`
`ClassDescriptor`	`addMultipleTablePrimaryKeyField`	`addForeignKeyFieldForMultipleTable`
`ClassDescriptor`	`addMultipleTablePrimaryKeyFieldName`	`addForeignKeyFieldNameForMultipleTable`
`ClassDescriptor`	`addMultipleTableForeignKeyFieldName`	`addForeignKeyFieldNameForMultipleTable`

Removed API

The following classes were removed in the release prior to TopLink 11*g* Release 1 (11.1.1):

- `OTSTransactionController`
- `OTSSynchronizationListener`
- `OracleSequenceDefinition` (use `SequenceObjectDefinition` instead)

- `TimeTenSequenceDefinition` (use `SequenceObjectDefinition` instead)

Miscellaneous API Changes

Other API changes include the following:

- JPA Persistence Provider Implementation.
- Session Finalizers Disabled by Default.
- Vector and Hashtable Return Types Changed to List or Map.

JPA Persistence Provider Implementation

The persistence provider implementation in all TopLink releases since 11g (11.1.1) is packaged in `eclipselink.jar`. It replaces all previous implementations, for example:

- `toplink.jar`
- `toplink-essentials.jar`

Session Finalizers Disabled by Default

In TopLink 11*g* (11.1.1) Technology Preview 3, session finalizers were disabled by default to improve performance. To enable session finalizers, use Session method `setIsFinalizersEnabled`(true).

Vector and Hashtable Return Types Changed to List or Map

Any `Session` or `ClassDescriptor` method that returns `Vector` or `Hashtable` will eventually be changed to return `List` or `Map`, respectively. To prepare for this change, cast `Vector` and `Hashtable` return types to `List` or `Map`, respectively. For example, although the Javadoc for `ClassDescriptor` method `getMappings` is `java.util.Vector`, you should cast the returned value to `List`:

```
List mappings = (List) descriptor.getMappings();
```

Other changes that now return `Map` include the following:

- `ClassDescriptor.getQueryKeys()`
- `ClassDescriptor.getProperties()`
- `DescriptorQueryManager.getQueries()`
- `EISInteraction.getProperties()`
- `Session.getProperties()`
- `Session.getQueries()`
- `getAttributesToAlwaysInclude()`
- `getSpecialOperations()`
- `getValuesToExclude()s`

Task 3: Rename Packages

EclipseLink continues to support native TopLink APIs; however, all `oracle.toplink. packages are now renamed to org.eclipse.persistence.`.

To migrate your application to use the new code base, you must rename the packages in your code. To facilitate this, a package renamer tool is included with the EclipseLink installation. Use this tool on all of the following:

- project source code
- `project.xml` file
- `persistence.xml` file
- `sessions.xml` file

The package renamer is located in the `*toplink_install_directory*``\toplink\utils\rename` directory. Windows and UNIX/LINUX scripts are included.

To run the package renamer using the scripts, do the following:

1. Find the `packageRename.cmd` (Windows) and `packageRename.sh` (UNIX/LINUX) scripts in `*toplink_install_directory*``\toplink\utils\rename` directory.
2. Run either `packageRename.cmd` or `packageRename.sh` with the following arguments:
   - `sourceLocation` - The directory containing the files to rename.
   - `targetLocation` - The destination directory for the renamed files. The package renamer removes any existing Java and XML files, so it is advisable to specify an empty directory.

     For example:

     ```
     packageRename c:/mySourceLocation c:/myDestinationLocation
     ```

     The package renamer performs a recursive directory search for Java and XML files to rename. The renamed version of each file is saved in the corresponding directory in the target location

Task 4: Convert XML Configuration Files

The package renamer can rename EclipseLink XML configuration files, but depending on the type of file, you may need to make additional changes.

Sessions XML

You can continue to use `sessions.xml` files as is. For a more forward-compatible solution, run the renamer on your `sessions.xml` files.

Deployment XML

Deployment XML files from TopLink 10.1.3 and above can be read by TopLink 11.1.1 and later. You

can continue to use those files or for a more forward compatible solution, run the renamer on these files and replace the version string in the "XML Header" with the following:

"Eclipse Persistence Services"

Persistence XML

To use EclipseLink as a persistence provider, you must run the renamer on your `persistence.xml` files. The renamer updates the persistence provider to be EclipseLink and also update any native TopLink specific properties to the EclipseLink equivalent.

ORM XML

The Object-Relational (ORM) XML configuration file (`orm.xml`) is not EclipseLink-dependant and does not need to be updated.

7.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

Chapter 8. Migrating from Hibernate to EclipseLink

This chapter describes how to migrate applications from using Hibernate JPA annotations and its native and proprietary API to using EclipseLink's JPA implementation, provided by EclipseLink. The migration involves converting Hibernate annotations to EclipseLink annotations, and converting native Hibernate API to EclipseLink JPA in the application code. Standard JPA annotations and API are left unchanged.

This chapter describes how to migrate applications from using Hibernate JPA annotations and its native and proprietary API to using EclipseLink JPA. The migration involves converting Hibernate annotations to EclipseLink annotations, and converting native Hibernate API to EclipseLink JPA in the application code. Standard JPA annotations and API are left unchanged.

This chapter includes the following sections:

Use Case

A developer wants to migrate applications using Hibernate as the persistence provider to use EclipseLink instead.

Solution

Follow the instructions in this chapter to upgrade the application.

Components

- EclipseLink 2.4 or later.

8.1. Introduction to the Solution

Hibernate is an object-relational mapping (ORM) 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`.

Reasons to Migrate

Reasons why you would want to migrate from Hibernate to EclipseLink include:

- **Performance and scalability**: EclipseLink's caching architecture allows you to minimize object creation and share instances. EclipseLink's caching supports single-node and clustered deployments.

- **Support for leading relation databases**: EclipseLink continues to support all leading relational databases with extensions specific to each. EclipseLink is also the best ORM solution for Oracle Database.
- **A comprehensive persistence solution**: While EclipseLink offers industry leading object-relational support, EclipseLink also uses its core mapping functionality to deliver Object-XML (JAXB), Service Data Object (SDO), and Database Web Services (DBWS). Depending on your requirements, you can use one or more of the persistence services based on the same core persistence engine.
- **JPA Support**: EclipseLink is the JPA reference implementation, and it will support future versions of JPA.

8.2. Main Tasks

Complete these tasks to migrate an application that uses Hibernate as its persistence provider to EclipseLink.

Task 1: Prerequisites

EclipseLink 2.4 or later.

Download EclipseLink from `http://www.eclipse.org/eclipselink/downloads/`.

Task 2: 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 8-1 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 8-1 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 EclipseLink handles selects, locks, polymorphism, and dynamic updates and inserts. For more information, see "EclipseLink/Examples/JPA/Migration/Hibernate/V3Annotations" in the Eclipselink documentation, at:

http://wiki.eclipse.org/EclipseLink/Examples/JPA/Migration/Hibernate/V3Annotations

Convert the SelectBeforeUpdate, dynamicInsert and dynamicUpdate Attributes

In Hibernate, the `selectBeforeUpdate` attribute specifies that Hibernate should never perform a SQL update unless it is certain that an object is actually modified. The `dynamicInsert` attribute specifies that the `INSERT SQL` statement should be generated at runtime and contain only the columns whose values are not null. The `dynamicUpdate` attribute specifies that the `UPDATE SQL` statement should be generated at runtime and can contain only those columns whose values have changed.

By default, EclipseLink will always insert all mapped columns and will update only the columns that have changed. If alternative operations are required, then the queries used for these operations can be customized by using Java code, SQL, or stored procedures.

Convert the OptimisticLock Attribute

In Hibernate, the `optimisticLock` attribute determines the optimistic locking strategy.

EclipseLink's optimistic locking functionality supports all of the Hibernate locking types and more. Table 8-1 translates locking types from Hibernate's `@Entity(optimisticLock)` attributes into EclipseLink locking policies. These policies can be configured either with the EclipseLink `@OptimisticLocking` annotation or in the EclipseLink `orm.xml` file. For more information, see @OptimisticLocking.

Table 8-1 Translating Hibernate's OptimisticLock to EclipseLink'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 the 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, EclipseLink 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.

Task 3: 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 8-2 illustrates a custom generator for sequence values.

Example 8-2 Custom Generator for Sequence Values

```
.
.
.
@Id
   @GeneratedValue(generator = "system-uuid")
   @GenericGenerator(name = "system-uuid", strategy = "mypackage.UUIDGenerator")
   public String getTransactionGuid()
.
.
.
```

In EclipseLink, a custom sequence generator can be implemented and registered by using the `@GeneratedValue` annotation. For more information, see "How to use Custom Sequencing" in the EclipseLink documentation, at:

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

Task 4: Convert Hibernate Mapping Annotations

The following sections describe how to convert various Hibernate annotations to EclipseLink annotations.

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.

EclipseLink does generate reasonable names for foreign keys, 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 EclipseLink 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.

The foreign key name is not used by EclipseLink 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.

Convert the @Cache Annotation

In Hibernate, the `@Cache` annotation configures the caching of entities and relationships. Because EclipseLink uses an entity cache instead of a data cache, the relationships are automatically cached. In these cases, the `@Cache` annotation should be removed during migration.When the `@Cache` annotation is used on an entity, its behavior is similar to the EclipseLink `@Cache` annotation. For more information about the `@Cache` annotation and equivalent `eclipselink-orm.xml` configuration values, see Jakarta Persistence API (JPA) Extensions Reference for EclipseLink.

Task 5: 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 8-3 illustrates a `persistence.xml` file for an application that uses Hibernate. Hibernate-specific values appear in bold font.

Example 8-3 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>
```

Modified persistence.xml File

Example 8-4 illustrates a `persistence.xml` file modified for an application that uses EclipseLink. Key differences include the value for the persistence provider. For EclipseLink, this value is `org.eclipse.persistence.jpa.PersistenceProvider`. The names of EclipseLink-specific properties are typically be prefixed by `eclipselink`, for example, `eclipselink.target-database`. EclipseLink-specific values appear in bold font.

Example 8-4 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>
    <!-- For Java SE applications, entity classes must be specified for EclipseLink
weaving. For Jakarta EE applications, the classes are found automatically. -->
    <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.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

Drop and Create the Database Tables

For production environments, you would usually have the schema set up on the database. The following properties defined in the persistence unit are more suitable for examples and demonstrations. These properties will instruct EclipseLink to automatically drop and create database tables. Any previously existing tables will be removed.

To use the Drop and Create Database Tables feature, add the following properties to the `persistence.xml` file.

```
<property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
 <property name="eclipselink.ddl-generation.output-mode" value="database"/>
```

For more information on this feature, see the `drop-and-create-tables` entry in "ddl-generation" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Create or Extend Database Tables

The Create or Extend Database Tables feature allows you match the database schema to the object model by creating new database tables or by modifying existing tables. You can modify existing tables by specifying field name changes and by add and removing fields.

In the current release, the Create or Extend Database Tables feature will not rename or delete existing columns. It will only add missing table columns.

The Create or Extend Database Tables feature reduces the need to repopulate test data. You avoid the need to use the Drop and Create Database Tables feature when the schema changes, due to changes in the object model. The Create or Extend Database Tables feature can also be used with extensibility to add table columns.

To use the Create or Extend Database Tables feature, add the following properties to the `persistence.xml` file. When the context is loaded, EclipseLink will query the database for each table

required in the persistence unit and use the results to determine if the table needs to be created or extended.

```
<property name="eclipselink.ddl-generation" value="create-or-extend-tables" />
 <property name="eclipselink.ddl-generation.output-mode" value="database" />
```

For more information on this feature, see the `create-or-extend-tables` entry in "ddl-generation" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Task 6: Convert Hibernate API to EclipseLink API

Table 8-2 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 `jakarta.persistence` package.

For information about the EclipseLink API, see *Java API Reference for EclipseLink*.

Table 8-2 Hibernate Classes and Equivalent JPA Interfaces

org.hibernate	jakarta.persistence	Description
`cfg.Configuration`	`Persistence`	Provides 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.

8.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- Hibernate at http://www.hibernate.org.
- "EclipseLink/Examples/JPA/Migration/Hibernate" in the EclipseLink documentation, at http://wiki.eclipse.org/EclipseLink/Examples/JPA/Migration/Hibernate.

Chapter 9. Using Multiple Databases with a Composite Persistence Unit

With EclipseLink, 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 this composite persistence unit are called *composite member persistence units*.

EclipseLink also supports multiple databases through partitioning. See Chapter 10, "Scaling Applications in Clusters" for more information.

This chapter includes the following sections:

Use Case

Users need to map expose multiple persistence units as a single persistence context within an application.

Solution

EclipseLink supports a "composite" persistence unit which can include multiple member persistence units.

Components

- EclipseLink 2.4.2 or later.
- Multiple databases.

Sample

See the following EclipseLink examples for related information:

- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/Composite`

9.1. Introduction to the Solution

With a composite persistence unit, you can:

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

Example 9-1 shows how you can persist data from a single persistence context into two different databases:

Example 9-1 Using Multiple Databases

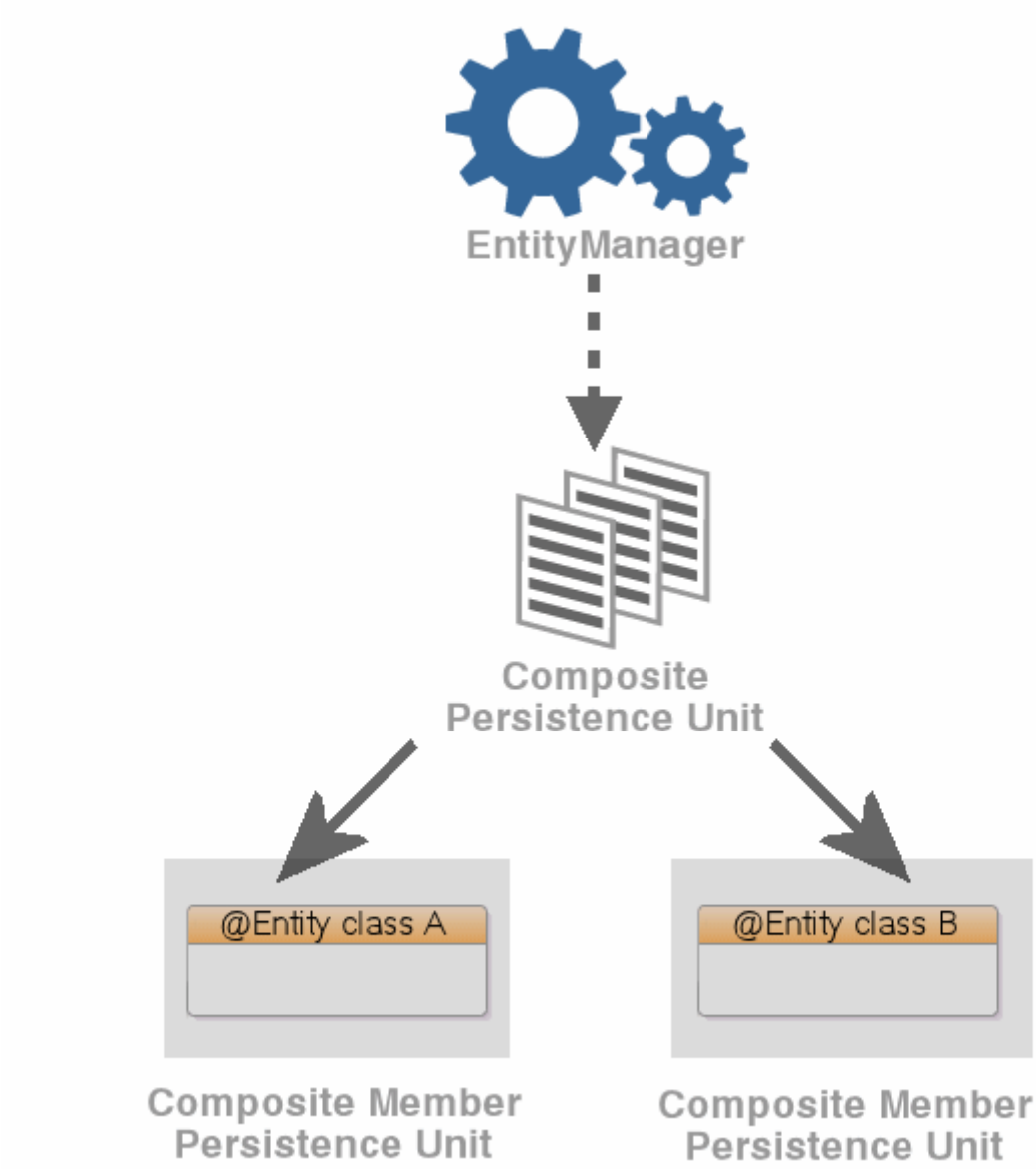
```
em.persist(new A(..));
em.persist(new B(..));
// You can insert A into database1 and insert B into database2.
// The two databases can be from different vendors.

em.flush();
```

Figure 9-1 illustrates a simple composite persistence unit. EclipseLink processes the `persistence.xml` file and detects the composite persistence unit, which contains two composite member persistence units:

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

Figure 9-1 A Simple Composite Persistence Unit



Description of "Figure 9-1 A Simple Composite Persistence Unit"

9.2. Composite Persistence Unit Requirements

When using composite persistence units, note 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 transaction types, target server information, and logging properties defined with composite members will be ignored.

9.3. Implementing the Solution

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

Task 1: Configure the Composite Persistence Unit

Because the composite persistence unit is a regular persistence element, it requires a `persistence.xml` file. Example 9-2 illustrates a sample `persistence.xml` file. Notice that there are no `datasource` or `jdbc` properties.

Example 9-2 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>
```

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

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

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 follows:

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

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

Or create it manually:

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

Task 3: Deploy Composite Persistence Units

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

- When deploying to Oracle WebLogic Server, package the JAR files in an EAR file or the `WEB-INF/lib` folder of a WAR file.
- When running as a standalone application, add the JAR files to the class path.

For important requirements, see Composite Persistence Unit Requirements.

9.4. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

For the following additional information about composite persistence units, see “@CompositeMember`,” "`composite.unit`," and "`composite-unit.member” in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*:

- Limitations of composite persistence units.
- Configuring composite member persistence units that contain dependencies.
- 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 the composite persistence unit `EntityManagerFactory`

Related Javadoc

For more information, see the following APIs in *Java API Reference for EclipseLink*:

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

Chapter 10. Scaling Applications in Clusters

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

This chapter includes the following sections:

Use Case

Applications must scale to meet client demand.

Solution

The implementation is achieved by using a cache, configuring cache coordination, and using data partitioning.

Components

- EclipseLink 2.4 or later.
- Application Server that supports clustering.
- Any compliant JDBC database.

Sample

See Additional Resources, for links to samples.

10.1. Introduction to the Solution

EclipseLink applications that are deployed to an application server cluster benefit from cluster scalability, load balancing, and failover. These capabilities ensure that EclipseLink applications are highly available and scale as application demand increases. EclipseLink applications are deployed the same way in application server clusters as they are in standalone server environments. However, additional planning and configuration is required to ensure cache consistency in an application server cluster.

EclipseLink uses 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 an application server cluster, caching can result in consistency issues (such as stale data) because changes made on one server are not reflected on objects cached in other servers. Cache consistency is problematic only for objects that are frequently updated. Read-only objects are not affected by cache consistency. For more details about caching, see:

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

Various options are available for addressing cache consistency:

- Use cache coordination. Cache coordination is a feature that broadcasts changes between the servers in the cluster to update or invalidate changed objects.
- Use distributed caching. EclipseLink Grid is an integration between EclipseLink and Oracle Coherence that addresses many cache consistency issues that result from operating in a distributed environment. For details on EclipseLink Grid, see
- Use optimistic locking. Optimistic locking is a feature that prevents updates to stale objects, and triggers the objects to be invalidated in the cache.
- Refresh the cache. Refreshing a cache loads that latest data in the cache.
- Disable the shared cache for highly volatile entities or limit the cache to read-only objects.

10.2. Implementing the Solution

These tasks provide general instructions for ensuring that a EclipseLink application scales in an application server cluster. Complete the tasks prior to deploying an application.

This section contains the following tasks:

- Task 1: Configure Cache Consistency
- Task 2: Ensure EclipseLink Is Enabled
- Task 3: Ensure All Application Servers Are Part of the Cluster
- Using Data Partitioning to Scale Data

Task 1: Configure Cache Consistency

This task includes different configuration options that mitigate the possibility that an application might use stale data when deployed to an application server cluster. The cache coordination option is specifically designed for applications that are clustered; 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 are the configuration options:

- Disabling Entity Caching
- Refreshing the Cache
- Setting Entity Caching Expiration
- Setting Optimistic Locking
- Using Cache Coordination

Oracle provides a EclipseLink and Coherence integration that allows EclipseLink to use Coherence as the L2 cache. For details on EclipseLink Grid, see

Disabling Entity Caching

Disable the shared cache for highly volatile entities or for all entities as required. 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>
```

The default configuration is `DISABLE_SELECTIVE` and allows caching to be disabled per entity. 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 {
}
```

Refreshing the Cache

Refreshing a cache reloads the cache from the database to ensure that an application is using current data. There are 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. The cache is only actually refreshed if the optimistic lock value in the database is newer than in the cache.

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

The `jakarta.persistence.Cache` interface includes methods that remove stale objects if the cache is out of date:

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

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

- The `evict` method invalidates specific classes.

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

The preceding methods are passive and refresh objects only 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.

Another possibility is to use the `setHint` method to set a query hint that triggers the query to refresh the cache. For example:

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

Lastly, native API methods are also available. For details, see the `ClassDescriptor` documentation in *Java API Reference for EclipseLink*.

Setting Entity Caching 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. There are 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 {
}
```

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. There are different ways to set optimistic locking.

The standard JPA `@Version` annotation is used for single valued value and timestamp based locking. However, for advanced locking features use the `@OptimisticLocking` annotation. 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 policies are available and are set within the `type` attribute:

- `ALL_COLUMNS`: This policy compares every field in the table in the `WHERE` clause when performing an update or delete operation.
- `CHANGED_COLUMNS`: This policy compares only the changed fields in the `WHERE` clause when performing an update operation. A delete operation compares only the primary key.
- `SELECTED_COLUMNS`: This policy compares selected fields in the `WHERE` clause when performing an update or delete operation. The fields that are 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.

Using Cache Coordination

Cache coordination synchronizes changes among distributed sessions. Cache coordination is most useful in application server clusters where maintaining consistent data for all applications is 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 (`EntityManagerFactory` or persistence unit) in the cluster. Cache coordination is most useful for applications 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 because of latency. In addition, cache coordination reduces the number of optimistic lock exceptions encountered in distributed architectures, and reduces 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 the Remote Method Invocation (RMI) and Java Message Service (JMS) protocols and is 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:

Jakarta Persistence API (JPA) Extensions Reference for EclipseLink

Setting Cache Synchronization

Cache synchronization determines how notifications of object changes are broadcast among session members. The following synchronization options are available:

- `SEND_OBJECT_CHANGES`: (Default) This option broadcasts a list of changed objects including data about the changes. This data is merged into the receiving cache.
- `INVALIDATE_CHANGED_OBJECTS`: This option broadcasts a list of the identities of the objects that

have changed. The receiving cache invalidates the objects rather than changing any of the data. This option is the lightest in terms of data sent and processing done in other cluster members.

- `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 synchronization. For example:

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

The `ClassDescriptor.setCacheSynchronizationType` native API method can also be used to specify synchronization. For details, see the `ClassDescriptor` documentation in *Java API Reference for EclipseLink*.

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. 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 because the topic provides 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 servers can look up each other. 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 usually not desirable. A second option is to set the URL programmatically using the cache coordination API. For more details, see "Configuring Cache Coordination Using the Cache Coordination API". 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 notification; asynchronous is the default. Synchronous broadcasting ensures that all of the servers are updated before a 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, then a separate channel can be used for each application. For example:

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

Last, 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" />
```

Cache Coordination and Oracle WebLogic

Both RMI and JMS cache coordination work with Oracle WebLogic Server. When a WebLogic cluster is used JNDI is replicated among the cluster servers, so a `cache.coordination.rmi.url` or a `cache.coordination.jms.host` option is not required. For JMS cache coordination, the JMS topic should only be deployed to only one of the servers (as of Oracle WebLogic 10.3.6). It may be desirable to have a dedicated JMS server if the JMS messaging traffic is heavy.

Use of other JMS services in WebLogic may have other requirements.

Cache Coordination and Glassfish

JMS cache coordination works with Glassfish Server. When a Glassfish cluster is used, JNDI is replicated among the cluster servers, so a `cache.coordination.jms.host` option is not required.

Use of other JMS services in Glassfish may have other requirements.

RMI cache coordination does not work when the JNDI naming service option is used in a Glassfish cluster. RMI will work if the `eclipselink.cache.coordination.naming-service` option is set to `rmi`. Each server must provide its own `eclipselink.cache.coordination.rmi.url` option, either by having a different `persistence.xml` file for each server, or by setting the URL as a System property in the server, or through a customizer.

Cache Coordination and IBM WebSphere

JMS cache coordination may have issues on IBM WebSphere. Use of a Message Driven Bean (MDB) may be required to allow access to JMS. To use an MDB with cache coordination, set the `eclipselink.cache.coordination.protocol` option to the value `jms-publishing`. The application will also have to deploy an MDB that processes cache coordination messages in its EAR file.

Configuring Cache Coordination Using the Cache Coordination API

The `CommandManager` interface allows you to programmatically configure cache coordination for a session. The interface is accessed using the `getCommandManager` method from the `DatabaseSession` interface.

Task 2: Ensure EclipseLink Is Enabled

Ensure that the EclipseLink JAR files are included on the classpath of each application server in the cluster to which the EclipseLink application is deployed and configure EclipseLink as the persistence provider. For detailed instructions about setting up EclipseLink with WebLogic Server and GlassFish Server, see Chapter 3, "Using EclipseLink with WebLogic Server," and Chapter 4, "Using EclipseLink with GlassFish Server," respectively.

Task 3: Ensure All Application Servers Are Part of the Cluster

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

TopLink relies on JMS and RMI and does not use the application server's cluster

communication.

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

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

10.3. Using Data Partitioning to Scale Data

Data partitioning allows an application to scale its data across more than one database machine. Data partitioning is supported at the entity level to allow a different set of entity instances for the same class to be stored in a different physical database or different node within a database cluster. Both regular databases and clustered databases are supported. Data can be partitioned both horizontally and vertically.

Partitioning can be enabled on an entity, a relationship, a query, or a persistence unit. To configure data partitioning, use the `@Partitioned` annotation and one or more partitioning policy annotations. Table 10-1 describes the partitioning policies

Table 10-1 Partitioning Policies

Annotation	Description
`@HashPartitioning`	Partitions access to a database cluster by the hash of a field value from the object, such as the object's ID, location, or tenant. The hash indexes into the list of connection pools/nodes. All write or read request for objects with that hash value are sent to the same server. If a query does not include the hash field as a parameter, it can be sent to all servers and unioned, or it can be left to the session's default behavior.
`@PinnedPartitioning`	Pins requests to a single connection pool/node. This allows for vertical partitioning.
`@RangePartitioning`	Partitions access to a database cluster by a field value from the object, such as the object's ID, location, or tenant. Each server is assigned a range of values. All write or read requests for objects with that value are sent to the same server. If a query does not include the field as a parameter, then it can either be sent to all servers and unioned, or left to the session's default behavior.

`@ReplicationPartitioning`	Sends requests to a set of connection pools/nodes. This policy is for replicating data across a cluster of database machines. Only modification queries are replicated.
`@RoundRobinPartitioning`	Sends requests in a round-robin fashion to the set of connection pools/nodes. This policy is used for load balancing read queries across a cluster of database machines. It requires that the full database be replicated on each machine, so it does not support partitioning. The data should either be read-only, or writes should be replicated.
`@UnionPartitioning`	Sends queries to all connection pools and unions the results. This is for queries or relationships that span partitions when partitioning is used, such as on a ManyToMany cross partition relationship.
`@ValuePartitioning`	Partitions access to a database cluster by a field value from the object, such as the object's location or tenant. Each value is assigned a specific server. All write or read requests for objects with that value are sent to the same server. If a query does not include the field as a parameter, then it can be sent to all servers and unioned, or it can be left to the session's default behavior.
`@Partitioning`	Partitions access to a database cluster by a custom partitioning policy. A class that extends the `PartitioningPolicy` class must be provided.

Partitioning policies are globally-named objects in a persistence unit and are reusable across multiple descriptors or queries. This improves the usability of the configuration, specifically with JPA annotations and XML.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. Connection pools are defined in the `persistence.xml` file for each participating database. Partition policies select the appropriate connection based on their particular algorithm.

If a transaction modifies data from multiple partitions, JTA should be used to ensure 2-phase commit of the data. An exclusive connection can also be configured in an `EntityManager` implementation to ensure only a single node is used for a single transaction.

The following example partitions the `Employee` data by location. The two primary sites, Ottawa and Toronto, are each stored on a separate database. All other locations are stored on the default database. Project is range partitioned by its ID. Each range of ID values are stored on a different database.

```
@Entity
@IdClass(EmployeePK.class)
@UnionPartitioning(
        name="UnionPartitioningAllNodes",
        replicateWrites=true)
@ValuePartitioning(
        name="ValuePartitioningByLOCATION",
        partitionColumn=@Column(name="LOCATION"),
        unionUnpartitionableQueries=true,
        defaultConnectionPool="default",
        partitions={
            @ValuePartition(connectionPool="node2", value="Ottawa"),
            @ValuePartition(connectionPool="node3", value="Toronto")
        })
@Partitioned("ValuePartitioningByLOCATION")
public class Employee {
    @Id
    @Column(name = "EMP_ID")
    private Integer id;
     @Id
    private String location;
    ...

    @ManyToMany(cascade = { PERSIST, MERGE })
    @Partitioned("UnionPartitioningAllNodes")
    private Collection<Project> projects;
    ...
}
```

The employee/project relationship is an example of a cross partition relationship. To allow the employees and projects to be stored on different databases a union policy is used and the join table is replicated to each database.

```
@Entity
@RangePartitioning(
        name="RangePartitioningByPROJ_ID",
        partitionColumn=@Column(name="PROJ_ID"),
        partitionValueType=Integer.class,
        unionUnpartitionableQueries=true,
        partitions={
            @RangePartition(connectionPool="default", startValue="0",
               endValue="1000"),
            @RangePartition(connectionPool="node2", startValue="1000",
               endValue="2000"),
            @RangePartition(connectionPool="node3", startValue="2000")
        })
@Partitioned("RangePartitioningByPROJ_ID")
public class Project {
    @Id
```

```
    @Column(name="PROJ_ID")
    private Integer id;
    ...
}
```

Clustered Databases and Oracle RAC

Some databases support clustering the database across multiple servers. Oracle Real Application Clusters (RAC) allows for a single database to span multiple different server nodes. Oracle RAC also supports table and node partitioning of data. A database cluster allows for any of the data to be accessed from any node in the cluster. However, it is generally more efficient to partition the data access to specific nodes, to reduce cross node communication. Partitioning can be used in conjunction with a clustered database to reduce cross node communication, and improve scalability. For details on using EclipseLink with Oracle RAC, see Using EclipseLink with Oracle RAC.

Adhere to the following requirements when using data partitioning with a database cluster:

- Partition policy should not enable replication, as database cluster makes data available to all nodes.
- Partition policy should not use unions, as database cluster returns the complete query result from any node.
- A `DataSource` and connection pool should be defined for each node in the cluster.
- The application's data access and data partitioning should be designed to have each transaction only require access to a single node.
- Usage of an exclusive connection for an `EntityManager` is recommended to avoid having multiple nodes in a single transaction and avoid 2-phase commit.

10.4. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

The following code sample and JavaDoc resources are available:

- Code Samples
  - `http://wiki.eclipse.org/EclipseLink/Examples/JPA/CacheCoordination`
  - `http://wiki.eclipse.org/EclipseLink/Examples/JPA/Caching`
- See the following APIs in *Java API Reference for EclipseLink*.
  - `org.eclipse.persistence.annotations.OptimisticLocking`
  - `org.eclipse.persistence.annotations.Cache`
  - `org.eclipse.persistence.annotations.Partitioned`
  - `org.eclipse.persistence.descriptors.ClassDescriptor`
  - `org.eclipse.persistence.sessions.coordination`

Chapter 11. Providing Software as a Service

This chapter introduces EclipseLink features available for developing shared applications that run in Software-as-a-Service (SaaS) environments.

Use Case

Users want to establish an SaaS environment, where applications and data are shared by multiple clients.

Solution

Use EclipseLink SaaS features, such as extensibility, client isolation, and external metadata sources.

Components

- EclipseLink 2.4 or later.

11.1. Introduction to the Solution

With EclipseLink, you can manage persistence in cloud-enabled applications and services. EclipseLink provides flexible SaaS solutions that address multi-tenancy and extensibility while still maintaining high performance and scalability, making the persistence layer of these applications a critical component.

These features are discussed in the following chapters:

- Chapter 12, "Making JPA Entities and JAXB Beans Extensible"
- Chapter 13, "Using an External MetaData Source"
- Chapter 14, "Tenant Isolation Using EclipseLink"

Chapter 12. Making JPA Entities and JAXB Beans Extensible

This chapter provides instructions for making JPA entities and JAXB beans extensible. Mappings can be added or modified externally, without modifying the entity or bean source file and without redeploying the persistence unit. This feature is useful in a Software-as-a-Service environment where multiple clients can share applications and datasources. It is also useful for customizing an application during installation rather than during development.

This chapter includes the following sections:

Use Case

Users want to establish a SaaS environment, where applications and datasources are shared by multiple clients.

Solution

Use the EclipseLink extensibility feature to extend JPA entities and JAXB beans by using external mappings.

Components

- EclipseLink 2.4 or later.

12.1. 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) architecture 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` interface to work with data after mappings have changed.
- Provide an additional source of metadata to be used by an application.

Main Tasks for Creating and Supporting an Extensible JPA Entity

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: Externalizing Extensions Using a MetaDataSource

Task 1: Configure the Entity

Configure the entity by annotating the entity class with `@VirtualAccessMethods` (or using the XML `<access-methods>`), adding `get` and `set` methods for the property values, and adding a data structure to store the extended attributes and values, as described in the following sections:

- Annotate the Entity Class with @VirtualAccessMethods
- Add get and set Methods to the Entity
- Define Virtual Attribute Storage
- Use XML

Annotate the Entity Class with @VirtualAccessMethods

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

Table 12-1 describes the attributes available to the `@VirtualAccessMethods` annotation.

Table 12-1 Attributes for the @VirtualAccessMethods Annotation

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

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.

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

Define Virtual Attribute Storage

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 3: Provide Additional Mappings.

A common way to store the virtual mappings is in a `Map` object (as shown in Example 12-1), but you can also use other strategies.

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 12-1 illustrates an entity class that uses property access.

Example 12-1 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);
    }
```

Use XML

As an alternative to, or in addition to, using the `@VirtualAccessMethods` annotation, you can use an

access="VIRTUAL" attribute on a mapping element (such as <basic>), for example:

```
<basic name="idNumber" access="VIRTUAL" attribute-type="String">
    <column name="FLEX_COL1"/>
</basic>
```

To set virtual access methods as the defaults for the persistence unit, use the <access> and <access-methods> elements, for example:

```
<persistence-unit-metadata>
           <xml-mapping-metadata-complete/>
           <exclude-default-mappings/>
           <persistence-unit-defaults>
                <access>VIRTUAL</access>
                <access-methods set-method="get" get-method="set"/>
              </persistence-unit-defaults>
   </persistence-unit-metadata>
```

Task 2: Design the Schema

Provide database tables with extra columns to store virtual attribute values. For example, the following Customer table includes two predefined columns, ID and NAME, and three columns for storing the attribute values, EXT_1, EXT_2, EXT_3:

CUSTOMER table

- INTEGER ID
- VARCHAR NAME
- VARCHAR EXT_1
- VARCHAR EXT_2
- VARCHAR EXT_3

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".

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" access="VIRTUAL" attribute-type="String">
  <column name="FLEX_COL1"/>
</basic>
```

Task 4: Externalizing Extensions Using a MetaDataSource

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 the `persistence.xml` file or by setting properties on the `EntityManagerFactory` interface, as described in the following sections.

For more information about external mappings, see Chapter 13, "Using an External MetaData Source."

Configure the persistence.xml File

In the `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"/>
```

Configure 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. EclipseLink 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 `EntityManagerFactory` interface 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 subclasses 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"/>
```

Refresh the Metadata Repository

If you change the metadata and you want an `EntityManager` instance based on the new metadata,

you must call the `refreshMetadata()` method on the `EntityManagerFactory` interface to refresh the data. The next `EntityManager` instance will be based on the new metadata.

The `refreshMetadata()` method takes a map of properties that can be used to override the properties previously defined for the `metadata-source` element.

Code Examples

Example 12-2 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`.

These items are illustrated in bold font.

Example 12-2 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 12-3 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`.
- The XML for extended mapping indicates which `get()` and `set()` method to use.

These items are illustrated in bold font.

Example 12-3 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"  access="VIRTUAL" attribute-type="String">
      <column name="FLEX_1"/>
    </basic>
```

Example 12-4 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`.

These items are illustrated in bold font.

Example 12-4 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);
    }

...
```

12.2. 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 extensions to JAXB introduce 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:

- Main Steps
- Code Examples

Main Steps

To create and support an extensible JAXB bean:

- Task 1: Configure the Bean

- Task 2: Provide Additional Mappings

Task 1: Configure the Bean

Configure the bean by 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. Alternatively, you can use the `<xml-virtual-access-methods>` element in `eclipselink-orm.xml`.

Annotate the Bean Class with @Xml VirtualAccessMethods

Annotate the bean with `@XmlVirtualAccessMethods` to specify that it is extensible and to define virtual properties.

Table 12-2 describes the attributes available to the `@XmlVirtualAccessMethods` annotation.

Table 12-2 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

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.

Define Virtual Attribute Storage

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.

Use XML

As an alternative to, or in addition to, using `@XmlVirtualAccessMethods`, you can use the XML equivalents, for example:

- 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"/>
```

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"/>
```

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.

Basic Setup

Example 12-5 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 12-5 A Base Class for Extensible Classes

```
package examples.virtual;

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

import jakarta.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 12-6 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 12-6 An Extensible Customer Class

```
package examples.virtual;

import jakarta.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 12-7 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 12-7 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 12-8 illustrates a `PhoneNumber` class. Like `Customer`, `PhoneNumber` will be an extensible class.

Example 12-8 An Extensible PhoneNumber Class

```
package examples.virtual;

import jakarta.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;
    }

}
```

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 captured in the `eclipselink-oxm.xml` mapping file or in files using the `eclipselink-orm.xml` schema.. 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 12-9 illustrates the `binding-tenant1.xml` mapping file.

Example 12-9 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 12-10 illustrates the `Customer` class code for tenant 1 to obtain the data associated with virtual properties.

Example 12-10 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 12-11 illustrates the XML output from the `Customer` class for tenant 1.

Example 12-11 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 12-12 illustrates the `binding-tenant2.xml` mapping file.

Example 12-12 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 12-13 illustrates the tenant 2 `Customer` class code to obtain the data associated with virtual properties.

Example 12-13 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 12-14 illustrates the XML output from the `Customer` class for tenant 2.

Example 12-14 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>
```

12.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- Code Sample
  - http://wiki.eclipse.org/EclipseLink/Examples/MySports
- “@VirtualAccessMethods” in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.)
- "Configuring Virtual Access Methods" in *Developing JAXB Applications Using EclipseLink MOXy*

Chapter 13. Using an External MetaData Source

This chapter provides instructions for storing mapping information in a metadata source that is external to the running application, so you can dynamically override or extend mappings in a deployed application.

This chapter includes the following sections:

Use Case

Users want to establish a SaaS environment, where applications are shared by multiple clients.

Solution

Employ EclipseLink SaaS features, such as extensibility, multi-tenancy, and external metadata sources.

Components

- EclipseLink 2.4 or later.

13.1. Introduction to the Solution

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.

13.2. Using the eclipselink-orm.xml File Externally

With EclipseLink, 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.

13.3. Main Tasks

To use an external metadata source for your mapping information, perform the following tasks:

Task 1: Configure the Persistence Unit

In your persistence unit, specify the external metadata source by defining an `eclipselink.metadata.source` property and assign as its value a class that implements `org.eclipse.persistence.jpa.metadata.MetadataSource`. For example:

```
<property name="eclipselink.metadata-source" value="mypackage.MyMetadataSource"/>
```

You are free to provide the metadata location in your class as you choose, for example:

```
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);
    }
}
```

Task 2: Configure the Server

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

- Static file serving
- A server-based solution with its own mapping file or a mapping file built on-demand from stored mapping information
- Some other web technology.

13.4. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter.

- For additional information about 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"
- For more information about persistence unit properties, see `PersistenceUnitProperties` class in *Oracle Fusion Middleware Java API Reference for EclipseLink*.
- For more information about the APIs, see the following in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*:

- “metadata-source”
- “metadata-source.properties.file”
- “metadata-source.send-refresh-command”
- “metadata-source.xml.file”
- “metadata-source.xml.url”

Chapter 14. Tenant Isolation Using EclipseLink

With EclipseLink, you can develop a single application and then deploy it for different clients, or "tenants," with varying degrees of application and data isolation and of tenant-specific functionality. For example, a large company may develop a single payroll application to be used by multiple divisions. Each division has access to its own data and to shared data, but they cannot see any other division's data.

This chapter includes the following sections:

Use Case

Multiple application clients must share data sources, with private access to their data, for example in a Software as a Service (SaaS) environment.

Solution

Decide on a strategy for tenant isolation; then use EclipseLink's tenant isolation features to implement the strategy.

Components

- EclipseLink 2.4 or later.
- A compliant Java Database Connectivity (JDBC) database, such as Oracle Database, Oracle Express, or MySQL.

14.1. Introduction to the Solution

EclipseLink offers considerable flexibility in how you can design and implement features for isolating tenants. Possibilities include the following:

Application Isolation options

- Separate container/server
- Separate application within the same container/server
- Separate entity manager factory and shared cache within the same application
- Shared entity manager factory with tenant isolation per entity manager

Data isolation options

- Separate database
- Separate schema/tablespace
- Separate tables
- Shared table with row isolation
- Query filtering
- Oracle Virtual Private Database (VPD)

EclipseLink includes the following options for providing multi-tenancy in the data source:

- **Single-table multi-tenancy** allows tenants to share tables. Each tenant has its own rows, identified by discriminator columns, and those rows are invisible to other tenants. See Using Single-Table Multi-Tenancy.
- With **table-per-tenant** multi-tenancy, each tenant has its own table or tables, identified by table tenant discriminators, and those tables are invisible to other users. See Using Table-Per-Tenant Multi-Tenancy.
- With (**VDP**) multi-tenancy, tenants use a VDP database, which provides the functionality to support multiple tenants sharing the same table. See Using VPD Multi-Tenancy.

EclipseLink further provides tenant-specific extensions through extensible entities using extensible entities and `MetadataSource`. For information about those features, see Chapter 12, "Making JPA Entities and JAXB Beans Extensible," and Chapter 13, "Using an External MetaData Source."

14.2. Using Single-Table Multi-Tenancy

With single-table multi-tenancy, any table (`Table` or `SecondaryTable`) to which an entity or mapped superclass maps can include rows for multiple tenants. Access to tenant-specific rows is restricted to the specified tenant.

Tenant-specific rows are associated with the tenant by using one or more tenant discriminator columns. Discriminator columns are used with application context values to limit what a persistence context can access.

The results of queries on the mapped tables are limited to the tenant discriminator value(s) provided as property values. This applies to all insert, update, and delete operations on the table. When multi-tenant metadata is applied at the mapped superclass level, it is applied to all subentities unless they specify their own multi-tenant metadata.

In the context of single-table multi-tenancy, ”single-table” means multiple tenants can share a single table, and each tenant’s data is distinguished from other tenants' data via the discriminator column(s). It is possible to use multiple tables with single-table multi-tenancy; but in that case, an entity’s persisted data is stored in multiple tables, and multiple tenants can share all the tables.

Main Tasks for Using Single-Table Multi-Tenancy

The following tasks provide instructions for using single-table multi-tenancy:

Task 1: Prerequisites

To implement and use single-table multi-tenancy, you need:

- EclipseLink 2.4 or later.

  Download EclipseLink from `http://www.eclipse.org/eclipselink/downloads/`.

- Any compliant Java Database Connectivity (JDBC) database, including Oracle Database, Oracle Database Express Edition (Oracle Database XE), or MySQL. These instructions are based on Oracle Database XE 11g Release 2.

  For the certification matrix, see

Task 2: Enable Single-Table Multi-Tenancy

Single-table multi-tenancy can be enabled declaratively using the `@Multitenant` annotation, in an Object Relational Mapping (ORM) XML file using the `<multitenant>` element, or by using annotations and XML together.

Using the @Multitenant Annotation

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

```
@Entity
@Table(name=EMP)
@Multitenant(SINGLE_TABLE)
public class Employee {
}
```

Single-table is the default multi-tenancy type, so `SINGLE_TABLE` does not have to be included in `@Multitenant`.

The `@Table` annotation is not required, because the discriminator column is assumed to be on the primary table. However, if the discriminator column is defined on a secondary table, you must identify that table using `@SecondaryTable`.

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>
```

Task 3: 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.

Tenant discriminator columns can be specified declaratively using the `@TenantDiscriminatorColumn` annotation or in an object-relational (ORM) XML file using the `<tenant-discriminator-column>` element.

The following characteristics apply to discriminator columns:

- Tenant discriminator column(s) must always be used with `@Multitenant` (or `<multitenant>` in the ORM XML file). You cannot specify the tenant discriminator column(s) only.
- The tenant discriminator column is assumed to be on the primary table unless another table is explicitly specified.
- On persist, the values of tenant discriminator columns are populated from their associated context properties.
- When a multi-tenant entity is specified, the tenant discriminator column can default. Its default values are:
  - Name = `TENANT_ID` (the database column name)
  - Context property = `eclipselink.tenant.id` (the context property used to populate the database column)
- Tenant discriminator columns are application definable. That is, the discriminator column is not tied to a specific column for each shared entity table. You can use `TENANT_ID`, `T_ID`, etc.
- There is no limit on the number of tenant discriminator columns an application can define.
- Any name can be used for a discriminator column.
- Generated schemas include specified tenant discriminator columns.
- Tenant discriminator columns can be mapped or unmapped:
  - When a tenant discriminator column is mapped, its associated mapping attribute must be marked as read only.
  - Both mapped and unmapped properties are used to form the additional criteria when issuing a SELECT query.

Use the @TenantDiscriminatorColumn Annotation

To use the `@TenantDiscriminatorColumn` annotation, include it with `@Multitenant` annotation on an entity or mapped superclass, and optionally include the `name` and `contextProperty` attributes. If you do not specify these attributes, the defaults `name = "TENANT-ID"` and `contextProperty = "eclipselink.tenant-id"` are used.

For example:

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

To specify multiple tenant discriminator 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() {
   ...
}
```

Use the <tenant-discriminator-column> Element

To use the `<tenant-discriminator-column>` element, include the element within a `<multitenant>` element and optionally include the `name` and `context-property` attributes. If you do not specify these attributes, the defaults `name = "TENANT-ID"` and `contextProperty = "eclipselink.tenant-id"` are used.

For example:

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

Map Tenant Discriminator Columns

Tenant discriminator columns can be mapped to a primary key or to 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() {
  ...
}
```

The following example uses the ORM XML file to map the tenant discriminator column to a primary key:

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

The following example maps the tenant discriminator column to another column named `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;
}
```

The following example uses the ORM XML file to map the tenant discriminator column to another column named `AGE`:

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

Define Persistence Unit and Entity Mappings Defaults

In addition to configuring discriminator columns at the entity and mapped superclass levels, you can also configure them at the `persistence-unit-defaults` and `entity-mappings` levels to provide defaults. Defining the metadata at the these levels follows similar JPA metadata defaulting and overriding rules.

Specify default tenant discriminator column metadata at the `persistence-unit-defaults` level in the ORM XML file. When defined at this level, the defaults apply to all entities of the persistence unit that have specified a multi-tenant type of `SINGLE_TABLE` minus those that specify their own tenant discriminator metadata. For example:

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <tenant-discriminator-column name="TENANT_ID" context-property="tenant.id"/>
  </persistence-unit-defaults>
```

```
</persistence-unit-metadata>
```

You can also specify tenant discriminator column metadata at the `entity-mappings` level in the ORM XML file. A setting at this level overrides a persistence unit default and applies to all entities with a multi-tenant type of `SINGLE_TABLE` of the mapping file, minus those that specify their own tenant discriminator metadata. For example:

```
<entity-mappings>
  ...
      ...
      <tenant-discriminator-column name="TENANT_ID" context-property="tenant.id"/>
      ...
</entity-mappings>
```

Configure Context Properties and Caching Scope

Runtime context properties are used in conjunction with the multi-tenancy configuration on an entity (or mapped superclass) to implement the multi-tenancy strategy. For example, the tenant ID assigned to a tenant discriminator column for an entity is used at runtime (via an entity manager) to restrict access to data, based on that tenant's ownership of (or access to) the rows and tables of the database.

At runtime, multi-tenancy properties can be specified in a persistence unit definition or passed to a create entity manager factory call.

The order of precedence for tenant discriminator column properties is as follows:

1. `EntityManager`
2. `EntityManagerFactory`
3. Application context (when in a Jakarta EE container)

For example, to set the configuration on a persistence unit in `persistence.xml`:

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

Alternatively, to set the properties programmatically:

```
HashMap properties = new HashMap();
properties.put("tenant.id", "707");
...
```

```
EntityManager em = Persistence.createEntityManagerFactory("multi-tenant",
      properties).createEntityManager();
```

Swapping tenant IDs during a live `EntityManager` is not allowed.

Configure a Shared Entity Manager

By default, tenants share the entity manager factory. A single application instance with a shared `EntityManagerFactory` for a persistence unit can be responsible for handling requests from multiple tenants.

The following example shows a shared entity manager factory configuration:

```
EntityManager em = createEntityManager(MULTI_TENANT_PU);
em.getTransaction().begin();
em.setProperty(EntityManagerProperties.MULTITENANT_PROPERTY_DEFAULT, "my_id");
```

When using a shared entity manager factory, the L2 cache is by default not shared, and therefore multi-tenant entities have an `ISOLATED` cache setting.

To share the cache, set the `eclipselink.multitenant.tenants-share-cache` property to `true`. This results in multi-tenant entities having a PROTECTED cache setting.

Queries that use the cache may return data from other tenants when using the `PROTECTED` setting.

Configure a Non-Shared Entity Manager

To create an entity manager factory that is not shared, set the `eclipselink.multitenant.tenants-share-emf` property to `false`.

When the entity manager factory is not shared, you must use the `eclipselink.session-name` property to provide a unique session name, as shown in the following example. This ensures that a unique server session and cache are provided for each tenant. This provides full caching capabilities. For example,

```
HashMap properties = new HashMap();
properties.put("tenant.id", "707");
properties.put("eclipselink.session-name", "multi-tenant-707");
...
EntityManager em = Persistence.createEntityManagerFactory("multitenant",
                     properties).createEntityManager();
```

Another example:

```
HashMap properties = new HashMap();
properties.put(PersistenceUnitProperties.MULTITENANT_SHARED_EMF, "false");
```

```
properties.put(PersistenceUnitProperties.SESSION_NAME, "non-shared-emf-for-this-emp");
properties.put(PersistenceUnitProperties.MULTITENANT_PROPERTY_DEFAULT, "this-emp");
...
EntityManager em = Persistence.createEntityManagerFactory("multi-tenant-pu",
properties).createEntityManager();
```

An example in the persistence unit definition:

```
<persistence-unit name="multi-tenant-pu">
  ...
  <properties>
    <property name="eclipselink.multitenant.tenants-share-emf" value="false"/>
    <property name="eclipselink.session-name"
         value="non-shared-emf-for-this-emp"/>
    <property name="eclipselink.tenant-id" value="this-emp"/>
    ...
  </properties>
</persistence-unit>
```

Configure an Entity Manager

When configuring properties at the level of the entity manager, you must specify the caching strategies, because the same server session can be used for each tenant. For example, you can set up an isolation level (L1 cache) to ensure no shared tenant information exists in the L2 cache. These settings are set when creating the entity manager factory.

```
HashMap tenantProperties = new HashMap();
properties.put("tenant.id", "707");

HashMap cacheProperties = new HashMap();
properties.put("eclipselink.cache.shared.Employee", "false");
properties.put("eclipselink.cache.size.Address", "10");
properties.put("eclipselink.cache.type.Contract", "NONE");
...
EntityManager em = Persistence.createEntityManagerFactory("multitenant",
                      cacheProperties).createEntityManager(tenantProperties);
...
```

Task 4: Perform Operations and Queries

The tenant discriminator column is used at runtime 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

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.

Task 5: Use Single-Table Multi-Tenancy in an Inheritance Hierarchy

Inheritance strategies are configured by specifying the inheritance type (`@jakarta.persistence.Inheritance`). Single-table multi-tenancy can be used in an inheritance hierarchy, as follows:

- Multi-tenant metadata can be applied only at the root level of the inheritance hierarchy when using a `SINGLE_TABLE` or `JOINED` inheritance strategy.
- You can also specify multi-tenant metadata within a `TABLE_PER_CLASS` inheritance hierarchy. In this case, every entity has its own table, with all its mapping data (which is not the case with `SINGLE_TABLE` or `JOINED` strategies). Consequently, in the `TABLE_PER_CLASS` strategy, some entities of the hierarchy may be multi-tenant, while others may not be. The other inheritance strategies can only specify multi-tenancy at the root level, because you cannot isolate an entity to a single table to build only its type.

14.3. Using Table-Per-Tenant Multi-Tenancy

Table-per-tenant multi-tenancy allows multiple tenants of an application to isolate their data in one or more tenant-specific tables. Multiple tenants' tables can be in a shared schema, identified using a prefix or suffix naming pattern; or they can be in separate, tenant-specific schemas. Table-per-tenant entities can be mixed with other multi-tenant type entities within the same persistence unit.

The table-per-tenant multi-tenant type is used in conjunction with:

- A tenant table discriminator that specifies the type of discriminator (schema or name with prefix or suffix)
- A tenant ID to identify the user (configured per entity manager or at the entity manager factory, if isolating the table-per-tenant per persistence unit.)

A single application instance with a shared `EntityManagerFactory` for a persistence unit can be responsible for handling requests from multiple tenants.

Alternatively, separate `EntityManagerFactory` instances can be used for each tenant. (This is required when using extensions per tenant.) In this case, tenant-specific schema and table names are defined in an `eclipselink-orm.xml` configuration file. A `MetadataSource` must be registered with a persistence unit. The `MetadataSource` is used to support additional persistence unit metadata provided from

outside the application.

For information about `MetadataSource`, see Chapter 13, "Using an External MetaData Source." See also `metadata-source` in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

The table-per-tenant multi-tenant type enables individual tenant table(s) to be used at the entity level. A tenant context property must be provided on each entity manager after a transaction has started.

- The table(s) (`Table` and `SecondaryTable`) for the entity are individual tenant tables based on the tenant context. Relationships within an entity that uses a join or a collection table are also assumed to exist within the table-per-tenant context.
- Multi-tenant metadata can only be applied at the root level of the inheritance hierarchy when using a `SINGLE_TABLE` or `JOINED` inheritance strategy. Multi-tenant metadata can be specified in a `TABLE_PER_CLASS` inheritance hierarchy

Main Tasks for Using Table-Per-Tenant Multi-Tenancy

The following tasks provide instructions for using table-per-tenant multi-tenancy:

- Task 1: Prerequisites
- Task 2: Enable Table-Per-Tenant Multi-Tenancy
- Task 3: Specify Tenant Table Discriminator
- Task 4: Specify a Context Property at Runtime

Task 1: Prerequisites

To implement and use table-per-tenant multi-tenancy, you need:

- EclipseLink 2.4 or later.

  Download EclipseLink from `http://www.eclipse.org/eclipselink/downloads/`.

- Any compliant Java Database Connectivity (JDBC) database, including Oracle Database, Oracle Database Express Edition (Oracle Database XE), or MySQL. These instructions are based on Oracle Database XE 11g Release 2.

  For the certification matrix, see

Task 2: Enable Table-Per-Tenant Multi-Tenancy

Table-per-tenant multi-tenancy can be enabled declaratively using the `@Multitenant` annotation; or in an Object Relational Mapping (ORM) XML file using the `<multitenant>` element, or using annotations and XML together.

Using the @Multitenant and @TenantTableDiscriminator Annotations

To use the `@Multitenant` annotation, include the annotation with an `@Entity` or `@MappedSuperclass` annotation and include the `TABLE_PER_TENANT` attribute.

For example:

```
@Entity
@Multitenant(TABLE_PER_TENANT
...)
public class Employee {
}
```

The `TABLE_PER_TENANT` attribute states that clients have a dedicated table or tables (`Table` and `SecondaryTable`) associated with the entity.

Using the <multitenant> Element

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

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

Task 3: Specify Tenant Table Discriminator

The tenant table discriminator describes the type of table discriminator to use in a table-per-tenant multi-tenancy strategy. The tenant table discriminator is identified by a property. You can define your own identifier or use the default property: `org.eclipse.persistence.config.PersistenceUnitProperties.MULTITENANT_PROPERTY_DEFAULT = "eclipselink.tenant-id"`

The tenant table discriminator can be specified at the entity or mapped superclass level, and it must always be accompanied with a `Multitenant(TABLE_PER_TENANT)` specification. It is not sufficient to specify only a tenant table discriminator.

The tenant table discriminator is used together with an associated application context to indicate which table or tables an application tenant can access.

Using the @TenantTableDiscriminator Annotation

Use the `@TenantTableDiscriminator` annotation to specify which tables are associated with which tenants. The tenant table discriminator must include a type and a context property:

- Use the `type` attribute to identify what type of discriminator to use:
  - Use `PREFIX` to apply the tenant table discriminator as a prefix to all multi-tenant tables.
  - Use `SUFFIX` to apply the tenant table discriminator as a suffix to all multi-tenant tables.
  - Use `SCHEMA` to apply the tenant table discriminator as a schema to all multi-tenant tables. This strategy requires appropriate database provisioning.

- Use the `contextProperty` attributes to identify the user. The value of the context property is a tenant ID that identifies the user. This can be configured for an entity manager or, if you want to isolate the table-per-tenant per persistence unit, an entity manager factory.

For example:

```
@Entity
@Table(name="EMP")
@Multitenant(TABLE_PER_TENANT)
@TenantTableDiscriminator(type=SCHEMA, contextProperty="eclipselink-tenant.id")
public class Employee {
    ...
}
```

Using the <tenant-table-discriminator> Element

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

```
<entity class="Employee">
  <multitenant type="TABLE_PER_TENANT">
    <tenant-table-discriminator type="SCHEMA"
        context-property="eclipselink-tenant.id"/>
  </multitenant>
  <table name="EMP">
  ...
</entity>
```

Task 4: Specify a Context Property at Runtime

At runtime, specify the context property using a persistence unit definition passed to an entity manager factory or set on an individual entity manager. For example:

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

To specify a context property at runtime programmatically:

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

```
    properties).createEntityManager();
```

An entity manager property definition follows:

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

Task 5: Perform Operations and Queries

The tenant discriminator column is used at runtime 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

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.

14.4. Using VPD Multi-Tenancy

A Virtual Private Database (VPD) uses security controls to restrict access to database objects based on various parameters.

For example, the Oracle Virtual Private Database supports security policies that control database access at the row and column level. Oracle VPD adds a dynamic `WHERE` clause to SQL statements issued against the table, view, or synonym to which the security policy was applied.

Oracle Virtual Private Database enforces security directly on the database tables, views, or synonyms. Because security policies are attached directly to these database objects, and the policies are automatically applied whenever a user accesses data, there is no way to bypass security.

When a user directly or indirectly accesses a table, view, or synonym that is protected with an

Oracle Virtual Private Database policy, Oracle Database dynamically modifies the SQL statement of the user. This modification creates a WHERE condition (called a predicate) returned by a function implementing the security policy. Oracle Virtual Private Database modifies the statement dynamically, transparently to the user, using any condition that can be expressed in or returned by a function. Oracle Virtual Private Database policies can be applied to SELECT, INSERT, UPDATE, INDEX, and DELETE statements.

When using EclipseLink VPD Multitenancy, the database handles the tenant filtering on all SELECT, INSERT, UPDATE, INDEX and DELETE queries.

To use EclipseLink VPD multi-tenancy, you must first configure VPD in the database and then specify multi-tenancy on the entity or mapped superclass, as shown in the following example, using `@Multitenant` and `@TenantDiscriminatorColumn`:

Main Tasks for Using VPD Multi-Tenancy

The following tasks provide instructions for using VPD multi-tenancy with Oracle Virtual Private Database:

- Task 1: Prerequisites
- Task 2: Configure the Virtual Private Database
- Task 3: Configure the Entity or Mapped Superclass
- Task 4: Disable Criteria Generation
- Task 5: Configure persistence.xml

Task 1: Prerequisites

To implement and use VPD multi-tenancy, you need:

- EclipseLink 2.4 or later.

  Download EclipseLink from `http://www.eclipse.org/eclipselink/downloads/`.

- Any compliant Java Database Connectivity (JDBC) database that supports VDP, for example, Oracle Virtual Private Database.

  For the certification matrix, see

Task 2: Configure the Virtual Private Database

In this example, an Oracle Virtual Private Database is configured with a policy and a stored procedure. The policy is a call to the database that tells the database to use a stored function to limit the results of a query. In this example, the function is called `ident_func`, and it is run whenever a `SELECT`, `UPDATE` or `DELETE` is performed on the `SCOTT.TASK` table. The policy is created as follows:

```
CALL DBMS_RLS.ADD_POLICY ('SCOTT', 'TASK', 'todo_list_policy', 'SCOTT', 'ident_func',
'select, update, delete'));
```

The function defined below is used by VPD to limit the data based on the identifier that is passed in to the connection. The function uses the `USER_ID` column in the table to limit the rows. The rows are limited, based on what is set in the `client_identifier` variable in the `userenv` context.

```
CREATE OR REPLACE FUNCTION ident_func (p_schema IN VARCHAR2 DEFAULT NULL, p_object IN
VARCHAR2 DEFAULT NULL)
    RETURN VARCHAR2
    AS
    BEGIN
       RETURN 'USER_ID = sys_context(''userenv'', ''client_identifier'')';
    END;
```

Task 3: Configure the Entity or Mapped Superclass

As described above, VPD is configured to use the `USER_ID` column to limit access to rows. Therefore, you must tell EclipseLink to auto-populate the `USER_ID` column on inserts. The following code uses EclipseLink multi-tenancy and specifies that the client identifier is passed in to the entity managers using a property called `tenant.id`. Because the filtering is done by VPD on the database, you must turn off caching on this entity to avoid leakage across users.

```
@Entity
@Multitenant(VPD)
@TenantDiscriminatorColumn(name = "USER_ID", contextProperty = "tenant.id")
@Cacheable(false)

public class Task implements Serializable {
...
...
```

Task 4: Disable Criteria Generation

When single-table and table-per-tenant multi-tenancy are enabled, a client identifier is auto appended to any generated SQL. However, when VPD is used to limit the access to data, the auto-appending of the identifier should be turned off.

Beginning with EclipseLink 2.4, disable criteria generation as follows:

```
@Multitenant(includeCriteria=false)
@TenantDiscriminatorColumn(name = "USER_ID", contextProperty = "tenant.id")
```

In EclipseLink 2.3.1, you must run the following codefrom a `SessionCustomizer`:

```
session.getDescriptor(Task.class).getQueryManager().setIncludeTenantCriteria(false);
```

Task 5: Configure persistence.xml

Add the following properties to `persistence.xml`.

Include the following to set and clear the VPD identifier:

```
<property name="eclipselink.session-event-listener"
value="example.VPDSessionEventAdapter" />
```

Include the following to provide one connection per entity manager:

```
<property name="eclipselink.jdbc.exclusive-connection.mode" value="Always" />
```

Include the following to allow native queries to be runnable from EclipseLink. This is required for creating VPD artifacts:

```
<property name="eclipselink.jdbc.allow-native-sql-queries" value="true" />
</properties>
```

For example:

```
 <properties>
   <property name="eclipselink.session-event-listener"
value="example.VPDSessionEventAdapter" />
   <property name="eclipselink.jdbc.exclusive-connection.mode" value="Always" />
   <property name="eclipselink.jdbc.allow-native-sql-queries" value="true" />
  ...
</properties>
```

14.5. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- Code Sample
  - http://wiki.eclipse.org/EclipseLink/Examples/MySports
- See the following in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.)
  - “@Multitenant”
  - “@TenantDiscriminatorColumn”
  - “@TenantDiscriminatorColumns”
  - “@TenantTableDiscriminator”

Chapter 15. Mapping JPA to XML

This chapter describes how to use JPA with the Java Architecture for XML Binding (JAXB)—the Jakarta EE standard for mapping POJOs (Plain Old Java Objects) to XML—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 includes the following topics:

Use Case

Users need to map JPA entities to XML.

Solution

EclipseLink provides support for the JAXB standard through EclipseLink MOXy extensions.

Components

- EclipseLink 2.4 or later.
- XML document

Sample

See the following EclipseLink and JAXB examples for related information:

- `http://wiki.eclipse.org/EclipseLink/Examples/MOXy`
- `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/index.html`

15.1. Introduction to the Solution

This chapter demonstrates some typical techniques for mapping JPA entities to XML. 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:

- Understanding MOXy
- Understanding an XML Data Representation

Understanding 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 relationships are traversed.

You can find examples of XML binding with JAXB in Binding JPA Entities to XML.

Understanding 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 leveraging 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 `jakarta.xml.bind.*` packages in the EclipseLink API. For more information about these annotations, see *Java API Reference for EclipseLink*.

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

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

Understanding MOXy

MOXy is EclipseLink's JAXB implementation. It allows you to map a 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 `jakarta.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 Xpath-based mapping, JSON binding, and 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 schema.
- It allows you to combine MOXy mappings and EclipseLink's persistence framework to interact with your data through JCA.
- It offers superior performance in several scenarios.

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

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

Understanding an XML Data Representation

Annotations are 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 tweak the metadata for future versions.
  - You use the standard JAXB annotations, and use the XML metadata for the MOXy extensions. In this way you don't 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.

To see how to use XML data representation, see Using XML Metadata Representation to Override JAXB Annotations

15.2. Binding JPA Entities to XML

The following examples demonstrate how to bind JPA entities to XML by using JAXB annotations. For more information about binding, see Understanding XML Binding for more information about JAXB, see Understanding JAXB

- Binding JPA Relationships to XML
- Binding Compound Primary Keys to XML
- Binding Embedded ID Classes to XML

Binding JPA Relationships to XML

The following exercise demonstrate show to use JAXB to derive an XML representation from a set of JPA entities, a process called "binding" (read about XML binding in Binding JPA Entities to XML). These examples will show how to bind two common JPA relationships:

- Privately-owned relationships
- Shared reference relationships

to map an Employee entity to that employee's phone number, address, and department.

Task 1: Define the Accessor Type and Import Classes

Since 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 classes:

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

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

Task 2: Map Privately-Owned Relationships

A "privately-owned" relationship occurs when the target object is only referenced 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 bi-directional mappings for both of these types of relationships between the `Employee` entity and the `Address` and `PhoneNumber` entities.

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 back. This is considered a one-to-one mapping because the employee can be associated with only one address. Since this relationship is bi-directional—that is, `Employee` points to `Address`, which must point back to `Employee`—it uses the EclipseLink 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 Task 1: Define the Accessor Type and Import Classes.
2. Map one direction of the relationship, in this case, the `employee` property on `Address`, 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 `Employee`—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 15-1 and Example 15-2.

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 phone 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 phone numbers and back. Since the relationship between `Employee` and `PhoneNumber` is bi-directional, the example again uses the EclipseLink 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 Task 1: Define the Accessor Type and Import Classes.
2. Map one direction of the relationship, in this case, the employee property on `PhoneNumber`, 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 phone number property on `Employee`—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 15-1 and Example 15-3.

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.

Since a shared reference relationship cannot be safely represented as nesting in XML, we use key relationships. In order to leverage the ID fields on JPA entities, you need to use the EclipseLink JAXB `@XmlID` annotation on non-String fields and properties and `@XmlIDREF` on string fields and properties.

This section contains examples that show how to map a many-to-one shared reference relationship and a many-to-many shared reference relationship.

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 phone 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 Task 1: Define the Accessor Type and Import Classes.
2. Map one direction of the relationship, in this case the phone number property on `Employee`, 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 PhoneNumber —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 15-1 and Example 15-3.

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. Since the relationship between `Department` and `Employee` is bi-directional, this example again uses the EclipseLink's `@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 Task 1: Define the Accessor Type and Import Classes.
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 association.

4. Complete the initial mapping—in this case, the `Department` property `employee`—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 Mapping a One-to-One and Embedded Relationship, specifying that `eId` is the primary key for JPA (`@Id` annotation), and for JAXB (`@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 15-1 and Example 15-4.

JPA Entities

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

- Example 15-1, "Employee Entity"
- Example 15-2, "Address Entity"
- Example 15-3, "PhoneNumber Entity"
- Example 15-4, "Department Entity"

In order to save space, package names, import statements, and the get/set methods have been omitted from the code examples. All examples use standard JPA annotations.

Example 15-1 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 15-2 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 15-3 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 15-4 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;

}
```

15.3. Binding Compound Primary Keys to XML

When a JPA entity has compound primary keys, you can bind it by using JAXB annotations and certain EclipseLink extensions, as shown in the following example.

Task1: Define the XML Accessor Type

Define the accessor type as `FIELD`, as described in Task 1: Define the Accessor Type and Import Classes

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— `eId`—can be annotated with the `@XmlID`.

   ```
       @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 doesn't 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 Example 15-5

Example 15-5 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;

}

public class EmployeeId {
    public BigDecimal eId;
    public String country;

    public EmployeeId(BigDecimal eId, String country) {
        this.id = id;
        this.country = country;;
    }

    public boolean equals(Object other) {
        if (other instanceof EmployeeId) {
```

```
            final EmployeeId otherEmployeeId = (EmployeeId) other;
            return (otherEmployeeId.eId.equals(eId) &&
otherEmployeeId.country.equals(country));
        }

    return false;
    }
}
```

Task 3: Create the Source Object

This Task creates the source object, the `PhoneNumber` entity. Because the target object has a compound key, we need to use the EclipseLink's `@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 EclipseLink's `@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 Example 15-6.

Example 15-6 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;

}
```

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. All attributes in the embedded Id's `Embeddable` are assumed to be part of the primary key. This exercise shows how to derive an XML representation from a set of JPA entities using JAXB when a JPA entity has an embedded ID class.

Task1: Define the XML Accessor Type

Define the XML accessor type as `FIELD`, as described in Task 1: Define the Accessor Type and Import Classes

Task 2: Create the Target Object

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

   ```
   @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 `PhoneNumber`. Because the relationship is bi-directional, use `@XmlInverseReference` 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 Example 15-7.

Example 15-7 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;

}
```

Task 3: Create the Source Object

The source object in this example has a compound key, so you must mark the field `@XmlTransient` to prevent a key from being mapped by itself. Use EclipseLink's `@XmlCustomizer` annotation to set up the mapping.

To create the source object, do the following:

1. Create the `PhoneNumber` entity.

```
@Entity
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 XML nodes for the mapping, using the EclipseLink `@XmlJoinNodes` annotation extension. If the target object had a *single* ID, you would use the `@XmlIDREF` annotation.

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

The completed `PhoneNumber` class should look like Example 15-8.

Example 15-8 PhoneNumber Class as Source Object

```
@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;

}
```

Task 5: Implement the DescriptorCustomizer as PhoneNumberCustomizer Class

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

1. Implement `DescriptorCustomizer` as `PhoneNumberCustomizer`. Be sure to 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()"`.

`PhoneNumberCustomizer` should look like Example 15-9.

Example 15-9 PhoneNumber Customizer with Updated Key Mappings

```
import org.eclipse.persistence.descriptors.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);
    }

}
```

Using the EclipseLink XML Binding Document

As demonstrated in the preceding examples, EclipseLink 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 Bindings documents.
- Specifying "virtual" mappings that do not correspond to concrete Java fields

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

XML_Bindings.

15.4. Mapping Simple Java Values to XML Text Nodes

This section demonstrates several ways to map simple Java values directly to XML text nodes. It includes the following examples:

- Mapping a Value to an Attribute
- Mapping a Value to a Text Node

Mapping a Value to an Attribute

This example 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 15-10.

Example 15-10 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, alternately, how to represent the value in EclipseLink's Object-to-XML Mapping (OXM) metadata format.

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 `jakarta.xml.bind.annotation.*`:

   ```
   package example;

   import jakarta.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 Example 15-11.

Example 15-11 Customer Object with Mapped id Property

```
package example;

import jakarta.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
   @XmlAttribute
   private Integer id;

   ...
}
```

Defining the Mapping in OXM Metadata Format

If you want to represent the mapping in EclipseLink's 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 15-12.

Example 15-12 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 Using XML Metadata Representation to Override JAXB Annotations.

Mapping a Value to a Text Node

EclipseLink 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 Sub-element
- Mapping Values to a Text Node by Position

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, alternately, by representing the mapping in EclipseLink's OXM metadata format.

Mapping by Using JAXB Annotations

Assuming the associated schema defines an element called `<phone-number>` which 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 `jakarta.xml.bind.annotation.*`:

   ```
   package example;

   import jakarta.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 Example 15-13.

Example 15-13 PhoneNumber Object with Mapped number Property

```
package example;
```

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

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

   ...
}
```

Defining the Mapping in OXM Metadata Format

If you want to represent the mapping in EclipseLink's 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 15-14.

Example 15-14 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>
...
```

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 EclipseLink's OXM metadata format. The following procedures illustrate how to map values for a customers' first names and last names

Mapping by Using JAXB Annotations

Assuming the associated schema defines the following elements:

- `<customer>` of the type customer-type, which itself is defined as a `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 name to the appropriate XML nodes. To do so:

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

```
package example;

import jakarta.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=`, 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 15-15.

Example 15-15 Customer Object Mapping Values to a Simple Sequence

```
package example;

import jakarta.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
   @XmlElement(name="first-name")
   private String firstName;

   @XmlElement(name="last-name")
   private String lastName;

   ...
}
```

Defining the Mapping in OXM Metadata Format

If you want to represent the mapping in EclipseLink's 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 15-16.

Example 15-16 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>
...
```

Mapping a Value to a Text Node in a Sub-element

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 EclipseLink's OXM metadata format. For example, if you want to populate `<first-name>` and `<last-name>` elements, which are sub-elements of a `<personal-info>` element under a `<customer>` root, you could use the following procedures to achieve these mappings.

Mapping by Using JAXB Annotations

Assuming the associated schema defines the following elements:

- `<customer>` of the type customer-type, which itself is defined as a complexTpe.
- `<personal-info>`
- Sub-elements 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 sub-element nodes. Because this example goes beyond a simple element name customization and actually introduces new XML structure, it uses EclipseLink's `@XmlPath` annotation. To achieve this mapping:

1. Create the object and import `jakarta.xml.bind.annotation.` and `org.eclipse.persistence.oxm.annotations.`.

   ```
   package example;

   import jakarta.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 sub-elements 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 Example 15-17.

Example 15-17 Customer Object Mapping Properties to Sub-elements

```
package example;

import jakarta.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;

   ...
}
```

Defining the Mapping in OXM Metadata Format

If you want to represent the mapping in EclipseLink's 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 15-18.

Example 15-18 Mapping Attributes as Sub-elements 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>
...
```

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 or by representing the mapping in EclipseLink's OXM metadata format. In the following example, XML contains two `<name>` elements; the first occurrence of name should represent the Customer's first name, the second name their last name.

Mapping by Using JAXB Annotations

Assuming an XML schema that defines the following attributes:

- `<customer>` of the type customer-type, which itself is specified as a `complexType`
- `<name>` of the type `String`

this example again 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 `jakarta.xml.bind.annotation.*` and `org.eclipse.persistence.oxm.annotations.XmlPath`.

   ```
   package example;

   import jakarta.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 Example 15-19.

Example 15-19 Customer Object Mapping Values by Position

```
package example;

import jakarta.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 about using XPath predicates, see Using XPath Predicates for Mapping.

15.5. Using XML Metadata Representation to Override JAXB Annotations

In addition to using Java annotations, EclipseLink 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 come from a third party).
- You do not want to introduce compile 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.

This section demonstrates how to use `eclipselink-oxm.xml` to override JAXB annotations

While using this mapping file enables many advanced features, it might prevent you from porting it to other JAXB implementations

Task 1: Define Advanced Mappings in the XML

First, update the XML mapping file to expose the `eclipselink_oxm_2_3.xsd`. schema. Example 15-20 shows how to modify the `<xml-bindings>` element in the mapping file to point to the correct namespace and leverage the schema. Each Java package can have one mapping file.

Example 15-20 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_4.xsd"
       version="2.4">
</xml-bindings>
```

Task 2: Configure Usage in JAXBContext

Next, pass the mapping file to `JAXBContext` in your object:

1. Specify the externalized metadata by inserting this code:

   ```
   Map<String, Object> properties = new HashMap<String, Object>(1);
   properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, "org/example/oxm.xml);
   JAXBContext.newInstance("org.example', aClassLoader, properties);
   ```

2. Create the properties object to pass to the `JAXBContext`. For this example:

   ```
   Map<String,Object> properties = new HashMap<String,Object>();
   properties.put(JAXBContextFactory.ECLIPSELINK_OXM_XML_KEY, metadata);
   ```

3. Create the `JAXBContext`. For this example:

```
JAXBContext.newInstance("example.order:example.customer", aClassLoader,
properties);
```

Task 3: Specify the 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:

```
jakarta.xml.bind.context.factory=org.eclipse.persistence.jaxb.JAXBContextFactory
```

2. Copy the `jaxb.properties` file to the package that contains your domain classes.

15.6. Using XPath Predicates for Mapping

This section demonstrates how the EclipseLink MOXy API uses XPath predicates to define an expression that specifiers 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:

- Understanding XPath Predicates
- Mapping Based on Position
- Mapping Based on an Attribute Value
- "Self" Mappings

Understanding XPath Predicates

As described above, an XPath predicate is an expression that defines a specific object-to-XML mapping when standard annotations

re not sufficient. For example, the following snippet of XML shows a `<data>` element with two `<node>` sub-elements. If you wanted to create this mapping in a Java object, you would need to specify an XPath predicate for each `<node>` sub-element; 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>
```

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.

Mapping Based on Position

This mapping technique is described in Mapping Values to a Text Node by Position.

Mapping Based on an Attribute Value

Beginning with EclipseLink MOXy 2.3, you can also map to an XML element based on an Attribute value. In this exercise, you will annotate the JPA entity to render the XML document shown in Example 15-21. Note that all of the XML elements are named node but are differentiated by the value of their name attribute.

Example 15-21

```
<?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, you need to 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.

Task 1: Create the Customer Entity

To create the `Customer` class entity:

1. Import the necessary JPA packages by adding the following code:

   ```
   import jakarta.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 local to the `Customer` class these properties:
   - `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 local to the `Customer` class, declare the `phoneNumber` property as a `List<PhoneNumber>` type and assign it the value `new ArrayList<PhoneNumber>()`.

The `Customer` class should look like the snippet in Example 15-22.

Example 15-22 Customer Object Mapping to an Attribute Value

```
package example;

import jakarta.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>();

    ...
```

```
}
```

Task 2: Create the Address Entity

To create the `Address` class, do the following:

1. Import the necessary JPA packages by adding the following code:

   ```
   import jakarta.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 the previous Tasks because the `Address` class is root not a root element in the XML document.

3. Declare local to the `Address` class the `String` property `street`. Set the XPath predicate by preceding the property declaration with the annotation `@XmlPath("node[@name='street']/text()")`.

The `Address` class should look like Example 15-23.

Example 15-23 Address Object Mapping to an Attribute Value

```
package example;

import jakarta.xml.bind.annotation.*;

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

@XmlAccessorType(XmlAccessType.FIELD)
public class Address {

    @XmlPath("node[@name='street']/text()")
    private String street;

    ...
}
```

Task 3: Create the PhoneNumber Entity

To create the `PhoneNumber` entity:

1. Import the necessary JPA packages by adding the following code:

```
import jakarta.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 Example 15-24.

Example 15-24 PhoneNumber Object Mapping to an Attribute Value

```
package example;

import jakarta.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {

    @XmlAttribute
    private String type;

    @XmlValue
    private String number;

    ...
}
```

"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 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 Task 1: Create the Customer Entity.
2. Declare local to the `Customer` class these properties:
   - `firstName` (String type)
   - `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 local to the `Customer` class, declare the `phoneNumber` property as a `List<PhoneNumber>` type and assign it the value `new ArrayList<PhoneNumber>()`.

The rendered XML for the Customer entity would look like Example 15-25.

Example 15-25 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>
```

15.7. Using Dynamic JAXB/MOXy

Dynamic JAXB/MOXy allows you to bootstrap a `JAXBContext` 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`.
- Dynamic entities offer a simple `get(propertyName)`/`set(propertyName propertyValue)` API to manipulate their data.
- Dynamic entities have an associated `DynamicType`, which is generated in-memory, when the metadata is parsed.

The following Tasks demonstrate how to use dynamic JAXB:

- 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

Task 1: Bootstrap a Dynamic JAXBContext from an XML Schema

This example demonstrates how to bootstrap a dynamic `JAXBContext` from an XML Schema.

Bootstrapping from an XML Schema

Use the `DynamicJAXBContextFactory` to create a dynamic `JAXBContext`. Example 15-26 to bootstrap a `DynamicJAXBContext` from the `customer.xsd` schema (Example 15-27) by using `createContextFromXSD()`.

Example 15-26 Specifying the Input Stream and Creating the 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`.

The XML Schema

Example 15-27 shows the `customer.xsd` schema that represents the metadata for the dynamic JAXBContext you are bootstrapping.

Example 15-27 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>
```

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`.

The following example shows two schema documents, `customer.xsd` (Example 15-28) and `address.xsd` Example 15-29). 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 15-28 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 15-29 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>
```

Implementing and Passing an EntityResolver

If you want to bootstrap `DynamicJAXBContext` from the `customer.xsd` schema, you need to pass an entity resolver. Do the following:

1. To resolve the locations of the imported schemas, you need to implement an `entityResolver` by supplying the code shown in Example 15-30.

   Example 15-30 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;
   }

}
```

1. After you implement your `DynamicJAXBContext`, pass the `EntityResolver`, as shown in Example 15-31.

   Example 15-31 Passing in the Entityresolver

```
FileInputStream xsdInputStream = new FileInputStream("src/example/customer.xsd");
DynamicJAXBContext jaxbContext =
    DynamicJAXBContextFactory.createContextFromXSD(xsdInputStream, new
MyEntityResolver(), null, null);
```

Error Handling

You might see the following exception when importing another schema:

```
Internal Exception: org.xml.sax.SAXParseException: schema_reference.4: Failed to read
schemadocument '<imported-schema-name>', because 1) could not find the document; 2)
the document couldnot be read; 3) the root element of the document is not
<xsd:schema>.
```

To work around this exception, disable XJC's schema correctness check by setting the `noCorrectnessCheck` Java property. You can set this property one of two ways:

- From within the code, by adding this line:

```
System.setProperty("com.sun.tools.xjc.api.impl.s2j.SchemaCompilerImpl.noCorrectness
Check", "true")
```

- From the command line, by using this command:

```
-Dcom.sun.tools.xjc.api.impl.s2j.SchemaCompilerImpl.noCorrectnessCheck=true
```

Specifying a ClassLoader

Use your application's current class loader as the `classLoader` parameter. This parameter verifies that specified classes exist before new `DynamicTypes` are generated. In most cases you can pass `null` for this parameter and use `Thread.currentThread().getContextClassLoader()` instead.

Task 2: Create Dynamic Entities and Marshal Them to XML

This example shows how to create dynamic entities and marshal then to XML.

Creating the Dynamic Entities

Use the `DynamicJAXBContext` to create instances of `DynamicEntity`. 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 15-32 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);
```

Marshalling the Dynamic Entities to XML

The marshaller obtained from the `DynamicJAXBContext` is a standard marshaller and can be used normally to marshal instances of DynamicEntity.

Example 15-33 Standard Dynamic JAXB Marshaller

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

Example 15-34 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>
```

Task 3: Unmarshal the Dynamic Entities from XML

In this example 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 15-34.

Unmarshal DynamicEntities from XML

The Unmarshaller obtained from the `DynamicJAXBContext` is a standard unmarshaller, and can be used normally to unmarshal instances of `DynamicEntity`.

Example 15-35 Standard Dynamic JAXB Unmarshaller

```
FileInputStream xmlInputStream = new
FileInputStream("src/example/dynamic/customer.xml");
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
DynamicEntity customer = (DynamicEntity) unmarshaller.unmarshal(xmlInputStream);
```

Get Data from the Dynamic Entity

Next, specify which data in the dynamic entity to obtain. Specify this value by using `System.out.println()` 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"));
```

Use DynamicType to Introspect Dynamic Entity

Instances of `DynamicEntity` have a corresponding `DynamicType`, which you can use to introspect the `DynamicEntity`, as shown in Example 15-36.

Example 15-36

```
DynamicType addressType = jaxbContext.getDynamicType("org.example.Address");

DynamicEntity address = customer.<DynamicEntity>get("address");
for(String propertyName: addressType.getPropertiesNames()) {
    System.out.println(address.get(propertyName));
}
```

15.8. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- *Developing JAXB Applications Using EclipseLink MOXy*

Chapter 16. Converting Objects to and from JSON Documents

This chapter describes how EclipseLink MOXy supports the ability to convert objects to and from JSON (JavaScript Object Notation). This feature is useful when creating RESTful services; JAX-RS services can accept both XML and JSON messages.

This chapter includes the following sections:

Use Case

Users need to convert objects to and from JSON documents.

Solution

EclipseLink provides JSON support through the EclipseLink MOXy implementation.

Components

- EclipseLink 2.4 or later.
- JSON documents.

Sample

See the following EclipseLink samples for related information:

- `http://wiki.eclipse.org/EclipseLink/Examples/MOXy`
- `http://wiki.eclipse.org/EclipseLink/Examples/MOXy/JSON_Metadata`
- `http://wiki.eclipse.org/EclipseLink/Examples/MOXy/MOXy_JSON_Provider`

16.1. Introduction to the Solution

EclipseLink supports all MOXy object-to-XML options when reading and writing JSON, including:

- EclipseLink's advanced and extended mapping features (in addition to the JAXB specification)
- Storing mappings in external bindings files
- Creating dynamic models with Dynamic JAXB
- Building extensible models that support multitenant applications

16.2. Implementing the Solution

This section contains the following tasks for converting objects to and from JSON documents.

Task 1: Marshalling and Unmarshalling JSON Documents

Use the **eclipselink.media-type** property on your JAXB Marshaller or Unmarsaller to produce and use JSON documents with your application, as shown in Example 16-1.

Example 16-1 Marshalling and Unmarshalling

```
...

Marshaller m = jaxbContext.createMarshaller();
m.setProperty("eclipselink.media-type", "application/json");

Unmarshaller u = jaxbContext.createUnmarshaller();
u.setProperty("eclipselink.media-type", "application/json");

...
```

You can also specify the `eclipselink.media-type` property in the `Map` of the properties used when you create the `JAXBContext`, as shown in Example 16-2.

Example 16-2 Using a Map

```
import org.eclipse.persistence.jaxb.JAXBContextProperties;
import org.eclipse.persistence.oxm.MediaType;

Map<String, Object> properties = new HashMap<String, Object>();
properties.put("eclipselink.media-type", "application/json");

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Employee.class }, properties);
Marshaller jsonMarshaller = ctx.createMarshaller();
```

```
Unmarshaller jsonUnmarshaller = ctx.createUnmarshaller();
```

When specified in a Map, the Marshallers and Unmarshallers created from the `JAXBContent` will automatically use the specified media type.

You can also configure your application to use JSON documents by using the `MarshallerProperties`, `UnmarshallerProperties`, and `MediaType` constants, as shown in Example 16-3.

Example 16-3 Using MarshallerProperties and UnarshallerProperties

```
import org.eclipse.persistence.jaxb.MarshallerProperties;
import org.eclipse.persistence.jaxb.UnarshallerProperties;
import org.eclipse.persistence.oxm.MediaType;

m.setProperty(MarshallerProperties.MEDIA_TYPE, MediaType.APPLICATION_JSON);
u.setProperty(UnmarshallerProperties.MEDIA_TYPE, MediaType.APPLICATION_JSON);
...
```

Task 2: Specifying JSON Bindings

Example 16-4 shows a basic JSON binding that does not require compile time dependencies in addition to those required for normal JAXB usage. This example shows how to unmarshal JSON from a `StreamSource` into the user object `SearchResults`, add a new `Result` to the collection, and then marshal the new collection to `System.out`.

Example 16-4 Using Basic JSON Binding

```
package org.example;

import org.example.model.Result;
import org.example.model.SearchResults;

import java.util.Date;

import jakarta.xml.bind.JAXBContext;
import jakarta.xml.bind.JAXBElement;
import jakarta.xml.bind.Marshaller;
import jakarta.xml.bind.Unmarshaller;
import javax.xml.transform.stream.StreamSource;

public class Demo {

    public static void main(String[] args) throws Exception {
        JAXBContext jc = JAXBContext.newInstance(SearchResults.class);

        Unmarshaller unmarshaller = jc.createUnmarshaller();
        unmarshaller.setProperty("eclipselink.media-type", "application/json");
        StreamSource source = new
```

```
StreamSource("http://search.twitter.com/search.json?q=jaxb");
        JAXBElement<SearchResults> jaxbElement = unmarshaller.unmarshal(source,
SearchResults.class);

        Result result = new Result();
        result.setCreatedAt(new Date());
        result.setFromUser("bsmith");
        result.setText("You can now use EclipseLink JAXB (MOXy) with JSON :)");
        jaxbElement.getValue().getResults().add(result);

        Marshaller marshaller = jc.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        marshaller.setProperty("eclipselink.media-type", "application/json");
        marshaller.marshal(jaxbElement, System.out);
    }

}
```

You can also write MOXy External Bindings files as JSON documents. Example 16-5 shows how to use `bindings.json` to map **Customer** and **PhoneNumber** classes to JSON.

Example 16-5 Using External Bindings

```
{
   "package-name" : "org.example",
   "xml-schema" : {
      "element-form-default" : "QUALIFIED",
      "namespace" : "http://www.example.com/customer"
   },
   "java-types" : {
      "java-type" : [ {
         "name" : "Customer",
         "xml-type" : {
            "prop-order" : "firstName lastName address phoneNumbers"
         },
         "xml-root-element" : {},
         "java-attributes" : {
            "xml-element" : [
                {"java-attribute" : "firstName","name" : "first-name"},
                {"java-attribute" : "lastName", "name" : "last-name"},
                {"java-attribute" : "phoneNumbers","name" : "phone-number"}
            ]
         }
      }, {
         "name" : "PhoneNumber",
         "java-attributes" : {
            "xml-attribute" : [
                {"java-attribute" : "type"}
            ],
            "xml-value" : [
```

```
                {"java-attribute" : "number"}
            ]
         }
      } ]
   }
}
```

Example 16-6 shows how to use the JSON file (created in Example 16-5) when bootstrapping a `JAXBContext`.

Example 16-6 Using JSON to Bootstrap a JAXBContext

```
Map<String, Object> properties = new HashMap<String, Object>(2);
properties.put("eclipselink.oxm.metadata-source", "org/example/binding.json");
properties.put("eclipselink.media-type", "application/json");
JAXBContext context = JAXBContext.newInstance("org.example",
Customer.class.getClassLoader() , properties);

Unmarshaller unmarshaller = context.createUnmarshaller();
StreamSource json = new StreamSource(new File("src/org/example/input.json"));
...
```

Task 3: Specifying JSON Data Types

Although XML has a single datatype, JSON differentiates between strings, numbers, and booleans. EclipseLink supports these datatypes automatically, as shown in Example 16-7

Example 16-7 Using JSON Data Types

```
public class Address {

   private int id;
   private String city;
   private boolean isMailingAddress;

}

{
   "id" : 1,
   "city" : "Ottawa",
   "isMailingAddress" : true
}
```

Task 4: Supporting Attributes

JSON does not use attributes; anything mapped with a `@XmlAttribute` annotation will be marshalled as an element. By default, EclipseLink triggers *both* the attribute and element events, thereby

allowing either the mapped attribute or element to handle the value.

You can override this behavior by using the `JSON_ATTRIBUTE_PREFIX` property to specify an attribute prefix, as shown in Example 16-8. EclipseLink prepends the prefix to the attribute name during marshal and will recognize it during unmarshal.

In the example below the `number` field is mapped as an attribute with the prefix @.

Example 16-8 Using a Prefix

```
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_ATTRIBUTE_PREFIX, "@");
jsonMarshaller.setProperty(MarshallerProperties.JSON_ATTRIBUTE_PREFIX, "@") ;
```

```
{
   "phone" : {
      "area-code" : "613",
      "@number" : "1234567"
   }
}
```

You can also set the `JSON_ATTRIBUTE_PREFIX` property in the Map used when creating the `JAXBContext`, as shown in Example 16-9. All marshallers and unmarshalers created from the context will use the specified prefix.

Example 16-9 Setting a Prefix in a Map

```
Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.JSON_ATTRIBUTE_PREFIX, "@");

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Phone.class }, properties);
```

Task 5: Supporting no Root Element

EclipseLink supports JSON documents without a root element. By default, if no `@XmlRootElement` annotation exists, the marshalled JSON document will not have a root element. You can override this behavior (that is omit the root element from the JSON output, even if the `@XmlRootElement` *is* specified) by setting the `JSON_INCLUDE_ROOT` property when marshalling a document, as shown in Example 16-10.

Example 16-10 Marshalling no Root Element Documents

```
marshaller.setProperty(MarshallerProperties.JSON_INCLUDE_ROOT, false);
```

When unmarshaling a document with no root elements, you should specify the class to which to unmarshal, as shown in Example 16-11.

Example 16-11 Unmarshalling no Root Element Documents

```
unmarshaller.setProperty(UnmarshallerProperties.JSON_INCLUDE_ROOT, false);
JAXBElement<SearchResults> jaxbElement = unmarshaller.unmarshal(source,
SearchResults.class);
```

If the document has no root element, you must specify the class to unmarshal to.

Task 6 Using Namespaces

Because JSON does not use namespces, by default all namespaces and prefixes are ignored when marshaling and unmarshaling. In some cases, this may be an issue if you have multiple mappings with the same local name – there will be no way to distinguish between the mappings.

With EclipseLink, you can supply a Map of namespace-to-prefix (or an instance of `NamespacePrefixMapper`) to the Marshaller and Unmarshaller. The namespace prefix will appear in the marshalled document prepended to the element name. EclipseLink will recognize the prefix during an unmarshal operation and the resulting Java objects will be placed in the proper namespaces.

Example 16-12 shows how to use the `NAMESPACE_PREFIX_MAPPER` property.

Example 16-12 Using Namesapces

```
Map<String, String> namespaces = new HashMap<String, String>();
namespaces.put("namespace1", "ns1");
namespaces.put("namespace2", "ns2");
jsonMarshaller.setProperty(MarshallerProperties.NAMESPACE_PREFIX_MAPPER, namespaces);
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_NAMESPACE_PREFIX_MAPPER,
namespaces);
```

The `MarshallerProperties.NAMESPACE_PREFIX_MAPPER` applies to *both* XML and JSON; `UnmarshallerProperties.JSON_NAMESPACE_PREFIX_MAPPER` is a *JSON-only* property. XML unmarshalling can obtain the namespace information directly from the document.

When JSON is marshalled, the namespaces will be given the prefix from the Map separated by a dot (`.`):

```
{
   "ns1.employee : {
      "ns2.id" : 123
   }
}
```

The dot separator can be set to any custom character by using the `JSON_NAMESPACE_SEPARATOR` property. Here, a colon (`:`) will be used instead:

```
jsonMarshaller.setProperty(MarshallerProperties.JSON_NAMESPACE_SEPARATOR, ':');
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_NAMESPACE_SEPARATOR, ':');
```

Task 7: Using Collections

By default, when marshalling to JSON, EclipseLink marshals empty collections as `[ ]`, as shown in Example 16-13.

Example 16-13 Marshalling Empty Collections

```
{
   "phone" : {
      "myList" : [ ]
   }
}
```

Use the `JSON_MARSHAL_EMPTY_COLLECTIONS` property to override this behavior (so that empty collections are not marshalled at all).

```
jsonMarshaller.setProperty(MarshallerProperties.JSON_MARSHAL_EMPTY_COLLECTIONS,
Boolean.FALSE) ;
```

```
{
   "phone" : {
   }
}
```

Task 8: Mapping Root-Level Collections

If you use the `@XmlRootElement(name="root")` annotation to specify a root level, the JSON document can be marshaled as:

```
marshaller.marshal(myListOfRoots, System.out);
```

```
[ {
   "root" : {
      "name" : "aaa"
   }
}, {
   "root" : {
      "name" : "bbb"
   }
} ]
```

Because the root element *is* present in the document, you can unmarsal it using:

```
unmarshaller.unmarshal(json);
```

If the class *does not* have an `@XmlRootElement` (or if `JSON_INCLUDE_ROOT` = false), the marshal would produce:

```
[ {
   "name":"aaa"
}, {
   "name":"bbb"
} ]
```

Because the root element *is not* present, you must indicate the class to unmarshal to:

```
unmarshaller.unmarshal(json, Root.class);
```

Task 9: Wrapping Text Values

JAXB supports one or more `@XmlAttributes` on `@XmlValue classes`, as shown in Example 16-14.

Example 16-14 Using @XmlAttributes

```
public class Phone {

   @XmlValue
   public String number;

   @XmlAttribute
   public String areaCode;

   public Phone() {
      this("", "");
   }

   public Phone(String num, String code) {
      this.number = num;
      this.areaCode = code;
   }

}
```

To produce a valid JSON document, EclipseLink uses a `value` wrapper, as shown in Example 16-15.

Example 16-15 Using a value Wrapper

```
{
   "employee" : {
      "name" : "Bob Smith",
      "mainPhone" : {
         "areaCode" : "613",
         "value" : "555-5555"
      },
      "otherPhones" : [ {
         "areaCode" : "613",
         "value" : "123-1234"
      }, {
         "areaCode" : "613",
         "value" : "345-3456"
      } ]
   }
}
```

By default, EclipseLink uses **value** as the name of the wrapper. Use the `JSON_VALUE_WRAPPER` property to customize the name of the value wrapper, as shown in Example 16-16.

Example 16-16 Customizing the Name of the Value Wrapper

```
jsonMarshaller.setProperty(MarshallerProperties.JSON_VALUE_WRAPPER, "$");
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_VALUE_WRAPPER, "$");
```

Would produce:

```
{
   "employee" : {
      "name" : "Bob Smith",
      "mainPhone" : {
         "areaCode" : "613",
         "$" : "555-5555"
      },
      "otherPhones" : [ {
         "areaCode" : "613",
         "$" : "123-1234"
      }, {
         "areaCode" : "613",
         "$" : "345-3456"
      } ]
   }
}
```

You can also specify the `JSON_VALUE_WRAPPER` property in the `Map` of the properties used when you create the `JAXBContext`, as shown in Example 16-17.

Example 16-17 Using a Map

```
Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.JSON_VALUE_WRAPPER, "$");

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Employee.class }, properties);
Marshaller jsonMarshaller = ctx.createMarshaller();
Unmarshaller jsonUnmarshaller = ctx.createUnmarshaller();
```

When specified in a Map, the Marshallers and Unmarshallers created from the `JAXBContent` will automatically use the specified value wrapper.

16.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- *Developing JAXB Applications Using EclipseLink MOXy*

Chapter 17. Testing JPA Outside a Container

This chapter describes how, with EclipseLink, you can use the persistence unit JAR file to test your application outside the container (for instance, in applications for the Java Platform, Standard Edition (Java SE platform)).

This chapter includes the following sections:

Use Case

Users need to use EclipseLink both inside and outside the container (such as applications for the Java SE platform).

Solution

This solution highlights the primary differences when using EclipseLink outside a container.

Components

- EclipseLink 2.4 or later.
- An application server (such as Oracle WebLogic Server, IBM WebSphere, or Glassfish)

17.1. Understanding JPA Deployment

When deploying outside of a container, use the `createEntityManagerFactory` method of the `jakarta.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.

This is a convenient way to set properties obtained from program input, such as the command line.

Using EntityManager

The `EntityManager` is the access point for persisting an entity bean, loading it from the database. Usually, the Jakarta Persistence API (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 Jakarta EE program container's data source. Because the managed data source is unavailable, you can pass properties to `createEntityManagerFactory` to change the transaction type from `JTA` to `RESOURCE_LOCAL` and to define JDBC connection information, as shown here:

Example 17-1 Changing transaction type and defining connection information

```
import static org.eclipse.persistence.jpa.config.PersistenceUnitProperties.*;

...

Map properties = new HashMap();

// Ensure RESOURCE_LOCAL transactions is used.
properties.put(TRANSACTION_TYPE,
  PersistenceUnitTransactionType.RESOURCE_LOCAL.name());

// Configure the internal EclipseLink connection pool
properties.put(JDBC_DRIVER, "oracle.jdbc.OracleDriver");
properties.put(JDBC_URL, "jdbc:oracle:thin:@localhost:1521:ORCL");
properties.put(JDBC_USER, "user-name");
properties.put(JDBC_PASSWORD, "password");
properties.put(JDBC_READ_CONNECTIONS_MIN, "1");
properties.put(JDBC_WRITE_CONNECTIONS_MIN, "1");

// Configure logging. FINE ensures all SQL is shown
properties.put(LOGGING_LEVEL, "FINE");

// Ensure that no server-platform is configured
properties.put(TARGET_SERVER, TargetServer.None);
```

You also have access to the EclipseLink extensions to the `EntityManager`.

17.2. Configuring the persistence.xml File

The `persistence.xml` file is the deployment descriptor file for persistence using JPA. It specifies the persistence units and declares the managed persistence classes, the object/relation mapping, and the database connection details.

Main Tasks

To configure the `persistence.xml` file, the following tasks:

- Task 1: Use the persistence.xml File
- Task 2: Instantiate EntityManagerFactory

Task 1: Use the persistence.xml File

Example 17-2 illustrates a `persistence.xml` file for a Java SE platform configuration (that is, outside a container).

Example 17-2 A persistence.xml File Specifying the Java SE Platform 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="jakarta.persistence.jdbc.driver"
value="oracle.jdbc.OracleDriver"/>
            <property name="jakarta.persistence.jdbc.url"
value="jdbc:oracle:thin:@localhost:1521:orcl"/>
            <property name="jakarta.persistence.jdbc.user" value="scott"/>
            <property name="jakarta.persistence.jdbc.password" value="tiger"/>
        </properties>
    </persistence-unit>
</persistence>
```

Task 2: Instantiate 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 17-2) by using:

```
Persistence.createEntityManagerFactory("my-app");
```

17.3. Using a Property Map

You can use a property map to override the default persistence properties and use container deployment.

Main Tasks

To use a property map, perform the following steps:

- Task 1: Configure the persistence.xml File
- Task 2: Configure the Bootstrapping API
- Task 3: Instantiate the EntityManagerFactory

Task 1: Configure the persistence.xml File

Example 17-3 illustrates a `persistence.xml` file that uses container deployment.

Example 17-3 A persistence.xml File Specifying the Java SE Platform Configuration, for use with a Property Map

```
<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>
```

There is no data source available when tested outside a container.

Task 2: Configure the Bootstrapping API

To test the persistence unit shown in Example 17-3 outside the container, you must use the Java SE platform bootstrapping API. Example 17-4 contains sample code that illustrates this bootstrapping.

Example 17-4 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);
```

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 17-4) by using:

```
Persistence.
createEntityManagerFactory("unitName", "properties");
```

17.4. Using Weaving

Weaving is a technique of manipulating the byte-code of compiled Java classes.

EclipseLink uses weaving to enhance Plain Old Java Object (POJO) classes and JPA entities with many features such lazy loading, change tracking, fetch groups, and internal optimizations.

How to Disable or Enable Weaving in a Java SE Environment

In a Java SE environment weaving is not enabled by default. This can affect LAZY One-To-One, Many-To-One and Basic relationships. It also has a major effect on performance and disable attribute change tracking.

To enable weaving in Java SE, the EclipseLink agent must be used when starting the Java VM.

```
java -javaagent:eclipselink.jar
```

Spring could also be used to allow JPA weaving in Java SE. See http://wiki.eclipse.org/EclipseLink/Examples/JPA/JPASpring for more information.

Static weaving can also be used, by including the following persistence property,

```
<property name="eclipselink.weaving" value="static"/>
```

See "weaving" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink* for more information.

How to Disable or Enable Weaving in a Jakarta EE Environment

In a Jakarta EE environment weaving is enabled by default (on any Jakarta EE 5 or greater fully compliant application server, such as Weblogic, Webspehere, and Glassfish. JBoss does not allow weaving so you must use static weaving or Spring).

To disable weaving the weaving persistence unit property can be used,

```
<property name="eclipselink.weaving" value="false">
```

For more information on weaving see "weaving" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

17.5. Additional Resources

For additional information about 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"

Related Javadoc

For more information, see the following APIs in *Java API Reference for EclipseLink*:

- `PersistenceUnitProperties` class
- `EntityManagerFactory` interface
- `JpaEntityManager` interface

Chapter 18. Enhancing Performance

This chapter describes EclipseLink performance features, provided by EclipseLink, and how to monitor and optimize EclipseLink-enabled applications.

This chapter includes the following sections:

- Performance Features
- Monitoring and Optimizing EclipseLink-Enabled Applications

Use Case

Users want to improve the performance of their EclipseLink-enabled application.

Solution

EclipseLink provides many configuration options that can improve performance, such as caching. In addition, there are ways to improve the performance of specific functions, such as using Join Fetching for queries.

Components

- EclipseLink 2.4 or later.

Sample

See the following EclipseLink samples for related information:

- `http://wiki.eclipse.org/EclipseLink/Performance`
- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/Performance`
- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/Monitoring`

18.1. Performance Features

EclipseLink 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
- Mapping
- Transactions
- Database
- Automated Tuning
- Tools

Object Caching

The EclipseLink 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 lives and the time of day, 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.

Caching Annotations

EclipseLink defines these entity caching annotations:

- `@Cache`
- `@TimeOfDay`
- `@ExistenceChecking`

EclipseLink also provides a number of persistence unit properties that you can specify to configure the EclipseLink cache (see "Persistence Property Extensions Reference" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*). These properties might compliment or provide an alternative to the usage of annotations.

Using the @Cache Annotation

EclipseLink 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 18-1 shows how to implement this annotation.

Example 18-1 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 "@Cache" in the *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Querying

The scope of a query, the amount of data returned, and how that data is returned can all affect the performance of a EclipseLink-enabled application. EclipseLink query mechanisms enhance query performance by providing these features:

- Read-only Queries
- Join Fetching
- Batch Reading
- Fetch Size
- Pagination
- Cache Usage

This section describes how these features improve performance.

Read-only Queries

EclipseLink uses the `eclipselink.read-only` hint, `QueryHint` (`@QueryHint`) to retrieve read-only results back from a query. On nontransactional 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 in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Join Fetching

Join Fetching 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 18-2. This example shows how the `@JoinFetch` annotation specifies the `Employee` field `managedEmployees`.

Example 18-2 Enabling JoinFetching

```
@Entity
 public class Employee implements Serializable {
     ...
     @OneToMany(cascade=ALL, mappedBy="owner")
     @JoinFetch(value=OUTER)
     public Collection<Employee> getManagedEmployees() {
         return managedEmployees;
     }
     ...
 }
```

For more details on Join Fetching, see "@JoinFetch" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Batch Reading

The `eclipselink.batch` hint supplies EclipseLink 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 only allowed on queries that have a single object in their select clause.

Fetch Size

If you have large queries that return a large number of objects you can improve performance by reducing the number database hits required to satisfy the selection criteria. To do this, use the 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, 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 if you are unsure of the result set size, incorrectly setting a fetch size too large or too small can decrease performance.

Pagination

Slow paging can result in significant application overhead; however, EclipseLink 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 EclipseLink 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 details on 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`

Cache Usage

EclipseLink 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 Java SE or Jakarta EE. Any entity persisted or removed using the entity manager will always be kept consistent with the cache.

You can specify how the query should interact with the EclipseLink cache by using the `eclipselink.cache-usage` hint. For more information, see "cache usage" in t*Jakarta Persistence API*

(JPA) Extensions Reference for EclipseLink.

Mapping

Mapping performance is enhanced by these features:

- Read-Only Objects
- Weaving

This section describes these features.

Read-Only Objects

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.

- To configure a read-only class for a single unit of work, specify that class as the argument to `addReadOnlyClass()`:

```
myUnitofWork.addReadOnlyClass(B.class);
```

- To configure multiple classes as read-only, add them to a vector and specify that vector as the argument to `addReadOnlyClass()`:

```
myUnitOfWork.addReadOnlyClasses(myVectorOfClasses);
```

For more information about using read-only objects to enhance performance, see "@ReadOnly" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Weaving

Weaving is a technique of manipulating the byte-code of compiled Java classes. The EclipseLink 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, EclipseLink uses dynamic weaving whenever possible. This includes inside an Jakarta EE 5/6 application server and in Java SE when the EclipseLink agent is configured. Dynamic weaving is recommended as it is easy to configure and does not require any changes to a project's build process

For details on how to use weaving to enhance application performance, see "weaving" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Transactions

To optimize performance during data transactions, use change tracking,. Change tracking allows

you to tune the way EclipseLink detects changes that occur during a transaction. You should choose the strategy based on the usage and data modification patterns of the entity type as 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 18-3.

Example 18-3 Enabling Change Tracking

```
@Entity
@Table(name="EMPLOYEE")
@ChangeTracking(OBJECT) (
public class Employee implements Serializable {
    ...
}
```

For more details on change tracking, see "@ChangeTracking" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Database

Database performance features in EclipseLink include:

- Connection Pooling
- Parameterized SQL and Statement Caching
- Batch Writing
- Serialized Object Policy

This section describes these features.

Connection Pooling

Establishing a connection to a data source can be time-consuming, so reusing such connections in a connection pool can improve performance. EclipseLink 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, EclipseLink 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. The also allow you to use 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 EclipseLink 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 (JTA).
- Default (write) and read connection pools;

- Sequence connection pools; Use these types of pools when your application requires table sequencing (that is, non-native 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 EclipseLink connection pools in a session.

For more information about using connection pools with EclipseLink, see the following topics in *EclipseLink Concepts*:

- "Understanding Connections"
- "Understanding Connection Pools"

Parameterized SQL and Statement Caching

Parameterized SQL can prevent the overall length of an 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, EclipseLink enables parameterized SQL but not prepared statement caching. You should enable statement caching either in EclipseLink when using an internal connection pool or in the data source when using an external connection pool and 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`.

EclipseLink determines binding behavior based on the database platform's support for binding. If the database does not support untyped parameter markers in specific SQL expressions, EclipseLink will disable parameter binding for the whole query.

To enable EclipseLink to restrict parameter binding decisions per expression, instead of per query, add this line to the persistence.xml file that is in the same path as your domain classes:

```
<property name="eclipselink.jdbc.allow-partial-bind-parameters" value="true"/>
```

For more information about using parameterized SQL and statement caching, see "jdbc.bind-parameters" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Batch Writing

Heterogeneous batch writing is an optimization that allows EclipseLink to send multiple heterogeneous dynamic SQL statements to the database to be executed as a single batch. Batch writing is best used for applications that perform multiples writes in each transaction.

To configure batch writing, include the `eclipselink.jdbc.batch-writing` and `eclipselink.jdbc.batch-writing.size` properties in the `persistence.xml` file. The following example enables Oracle's native batch writing feature that is available with the Oracle JDBC driver and configures the batch size to 150 statements:

```
<property name="eclipselink.jdbc.batch-writing" value="Oracle-JDBC"/>
<property name="eclipselink.jdbc.batch-writing.size" value="150"/>
```

Different batch options are supported and custom batch implementations can also be used. For a detailed reference of the batch writing properties, see the `batch-writing` and `batch-writing.size` documentation in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Serialized Object Policy

Serialized object policy is an optimization that allows EclipseLink to write out the whole entity object with its privately owned (and nested privately owned) entities and element collections into an additional field in the database. Serialized object policy optimizes fetching from the database, provides faster database reads, and reduces middle tier CPU and network access in certain situations.

Serialized object policy is best for read-only or read-mostly applications and should only be used for entities that load all their dependent entities or element collections. When using serialized object policy, database write operations (insert and update) are slower and queries for objects without private-owned data are slower. See "A Simple Serialized Object Policy Example" that demonstrates when serialized object policy is best used to increase performance.

Consider using serialized object policy only for complex objects with numerous aggregation as characterized by:

- Multiple database rows mapped to a single Java object
- When the object is read from the database all these rows are read at once (no indirection, or all indirection always triggered). There may be un-triggered indirection for other fields that are not included in the serialized object policy field
- If versionning is used, then updating or deleting any mapped row (or inserting of a new one) should result in incrementing of the object's version
- Object deletion causes all the rows to be deleted.
- Irregular structure of the aggregation makes it less possible to use other common optimizations (such as join fetching and batch reading).

Serialized Object Policy Configuration

Serialized object policy is enabled by using the `@SerializedObject` annotation on an entity or mapped superclass and passing in an implementation of the `SerializedObjectPolicy` interface. You must provide an implementation of this interface; there is no default implementation. The annotations also includes a field to define the column name for the object in the database. The default column name is `SOP`.

Example 18-4 enables serialized object policy, overrides the default column name, and sets optimistic locking to `cascade`, which can increase performance by keeping the serialized object policy field in the database up-to-date.

If serialized object policy is set on an entity, then policies with the same fields are set on all inheriting entities.

Example 18-4 Enabling Serialized Object Policy Using Annotations

```
@Entity
@SerializedObject(MySerializedObjectPolicy.class)
@OptimisticLocking(cascade = true)
public class Employee implements Serializable {
...

@Entity
@SerializedObject(MySerializedObjectPolicy.class, column = @Column(name="ADDR_SOP"))
@OptimisticLocking(cascade = true)
public class Address implements Serializable {
...
```

Example 18-5 enables serialized object policy in the `eclipselink-orm.xml` file

Example 18-5 Enabling Serialized Object Policy Using eclipselink-orm.xml

```
<entity class="Employee">
    <optimistic-locking cascade="true">
    <serialized-object class="MySerializedObjectPolicy">
</entity>

<entity class="Address">
    <optimistic-locking cascade="true">
    <serialized-object class="MySerializedObjectPolicy">
        <column name="ADDR_SOP"/>
    </serialized-object>
</entity>
```

Example 18-6 enables serialized object policy in a customizer (either session or descriptor):

Example 18-6 Enabling Serialized Object Policy in a Customizer

```
if (descriptor.hasSerializedObjectPolicy()) {

    MySerializedObjectPolicy sop = (MySerializedObjectPolicy)descriptor.
       getSerializedObjectPolicy();

    // to compare pk cached in SOP Object with pk read directly from the row from
    //pk field(s) (false by default):
```

```
    sop.setShouldVerifyPrimaryKey(true);

    // to NOT compare version cached in SOP Object with version read directly from
    // the row from version field (true by default):

    sop.setShouldVerifyVersion(false);

    // to define recoverable SOP (false by default):

    sop.setIsRecoverable(true);
}
```

To use a descriptor customizer, define the class and specify it using the `@Customizer` annotation:

```
public class MyDescriptorCustomizer implements
   org.eclipse.persistence.config.DescriptorCustomizer {
    public void customize(ClassDescriptor descriptor) throws Exception
    {
         ...
    }
}
...
@Customizer(MyDescriptorCustomizer.class)
public class Employee implements Serializable {...
```

To use a session customizer to reach all descriptors at once, specify it in a persistence unit property:

```
public class MySessionCustomizer implements
   org.eclipse.persistence.config.SessionCustomizer {
    public void customize(Session session) throws Exception
    {
        for (ClassDescriptor descriptor : session.getDescriptors().values()) {
            ...
        }
    }
}

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

Read queries (including find and refresh) automatically use a serialized object if serialized object policy is enabled. If the serialized object column contains `null`, or an obsolete version of the object, then a query using a serialized object policy would either throw an exception or, if all other fields have been read as well, build the object using these fields (exactly as in the case where a serialized object policy is not used).

To disable querying the serialized object, set the `SERIALIZED_OBJECT` property to `false` as part of a query hint. For example:

```
Query query = em.createQuery("SELECT e FROM Employee e")
   .setHint(QueryHints.SERIALIZED_OBJECT, "false");
```

The following example demonstrates disabling searching for a serialized object:

```
Map hints = new HashMap();
hints.put("eclipselink.serialized-object", "false");
Employee emp = em.find(Employee.class, id, hints);
```

Applications that use serialized object policy should also consider using the result set access optimization. Use the optimization when querying to avoid the costly reading of the serialized object policy field (which can be large) if it is already cached and the query is not a refresh query. The optimization ensures that only the primary key is retrieved from the result set and only gets additional values if the cached object cannot be used. To enable the result set access optimization, set the `eclipselink.jdbc.result-set-access-optimization` persistent unit property to `true` in the `persistence.xml` file. For example:

```
<property name="eclipselink.jdbc.result-set-access-optimization" value="true"/>
```

A Simple Serialized Object Policy Example

Consider the following example object model:

```
@Entity(name="SOP_PartOrWhole")
@Table(name="SOP_PART_OR_WHOLE")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@Index(columnNames={"LEFTPART_ID", "RIGHTPART_ID"})
public abstract class PartOrWhole implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    public long id;

    protected String description = "";

    @OneToOne(cascade=CascadeType.ALL, orphanRemoval=true)
    protected Part leftPart;
    @OneToOne(cascade=CascadeType.ALL, orphanRemoval=true)
    protected Part rightPart;
}

@Entity(name="SOP_Whole")
@DiscriminatorValue("W")
@SerializedObject(MySerializedObjectPolicy.class)
@NamedQueries({
    @NamedQuery(name="findWhole", query="Select w from SOP_Whole w where w.id =
    :id", hints= @QueryHint(name="eclipselink.serialized-object", value="false")),
```

```
    @NamedQuery(name="findWholeSOP", query="Select w from SOP_Whole w where w.id =
    :id"),
})
public class Whole extends PartOrWhole {
}

@Entity(name="SOP_Part")
@DiscriminatorValue("P")
public class Part extends PartOrWhole {
}
```

The above data model allows the construction of a `Whole` object with any number of (nested) `Part` objects. For example:

- 1 level – A `Whole` object contains left and right `Part` objects (3 objects all together)
- 2 levels – A `Whole` object contains left and right `Part` objects; each of the `Part` objects has left and right `Part` objects (7 objects all together)
- 3 levels – A `Whole` object contains left and right `Part` object; each of the `Part` objects has a left and right `Part` objects; which each have a left and right `Part` objects (15 objects all together)
- `` `n` `` levels – (2 `` `n` ``+1 - 1 objects all together)

Performance for the above data model increases as the number of levels in the model increases. For example:

- 1 level – performance is slower than without serialized object policy.
- 2 levels – performance is only slightly faster than without serialized object policy.
- 5 levels – performance is 7 times faster than without serialized object policy.
- 10 levels – performance is more than 25 times faster than without serialized object policy.

Automated Tuning

Automated tuning is an optimization that allows applications to automatically tune JPA and session configuration for a specific purpose. Multiple configuration options can be configured by a single tuner and different configurations can be specified before and after application deployment and after application metadata has been processed but before connecting the session. Automated tuning simplifies configuration and allows a dynamic single tuning option.

Tuners are created by implementing the `org.eclipse.persistence.tools.tuning.SessionTuner` interface. Two tuner implementations are provided and custom tuners can be created as required:

- Standard (`StandardTuner`) – The standard tuner is enabled by default and does not change any of the default configuration settings.
- Safe (`SafeModeTuner`) – The safe tuner configures the persistence unit for debugging. It disables caching and several performance optimizations to provide a simplified debugging and development configuration:

```
WEAVING_INTERNAL = false
WEAVING_CHANGE_TRACKING = false
CACHE_SHARED_DEFAULT = false
JDBC_BIND_PARAMETERS = false
ORM_SCHEMA_VALIDATION = true
TEMPORAL_MUTABLE = true
ORDER_UPDATES = true
```

To enable a tuner, specify a predefined tuner or enter the fully qualified name of a `SessionTuner` implementation as the value of the `eclipselink.tuning` property in the `persistence.xml` file. The following example enables the safe tuner.

```
<property name="eclipselink.tuning" value="Safe"/>
```

For a detailed reference of the `tuning` property, see *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Tools

EclipseLink provides monitoring and optimization tools, as described in Monitoring and Optimizing EclipseLink-Enabled Applications.

18.2. Monitoring and Optimizing EclipseLink-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. This section contains information about these subjects:

- Performance Optimization Recommendations and Tips
- Task 1: Measure EclipseLink Performance with the EclipseLink Profiler
- Task 2: Measure EclipseLink Performance in the Server Environment
- Task 3: Measure Fetch Group Field Usage
- Task 4: Identify Sources of Application Performance Problems
- Task 5: Modify Poorly-Performing Application Components
- Task 6: Measure Performance Again

Performance Optimization Recommendations and Tips

EclipseLink 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.

The most important concept associated with tuning your EclipseLink 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 EclipseLink Performance with the EclipseLink Profiler.
- Task 2: Measure EclipseLink Performance in the Server Environment
- Task 3: Measure Fetch Group Field Usage
- Task 4: Identify Sources of Application Performance Problems.
- Task 5: Modify Poorly-Performing Application Components.
- Task 6: Measure Performance Again.

Task 1: Measure EclipseLink Performance with the EclipseLink Profiler

The EclipseLink performance profiler helps you identify performance problems by logging performance statistics for every executed query in a given session. Use the performance profiler to monitor a single query, or simple single-threaded use case.

The EclipseLink performance profiler logs the following information to the log file.

Table 18-1 Information Logged by the EclipseLink 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)
SQL Generation	the amount of time spent to generate the SQL script before it is sent to the database (in milliseconds)

Enabling the EclipseLink Profiler

The EclipseLink performance profiler is an instance of `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 EclipseLink profiler, The `PerformanceProfiler` class public API also provides the functionality described in Table 18-2:

Table 18-2 Additional PerformanceProfiler Functionality

To...	**Use...**
Disable the profiler	`dontLogProfile`
Organize the profiler log into a summary of all the individual operation profiles including operation statistics like the shortest time of all the operations that were profiled, the total time of all the operations, the number of objects returned by profiled queries, and the 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`

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 EclipseLink-enabled application.

Example 18-7 shows an sample of the EclipseLink profiler output.

Example 18-7 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 18-7 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.

Task 2: Measure EclipseLink Performance in the Server Environment

Use the Performance Monitor to provide detailed profiling and monitoring information in a multithreaded server environment. Use the performance monitor to monitor a server, multiple threads, or long running processes.

Enable the monitor in `persistence.xml` file as follows:

```
<property name="eclipselink.profiler" value="PerformanceMonitor"/>
```

The performance monitor can also be enabled through code using a `SessionCustomizer`.

The performance monitor will output a dump of cumulative statistics every minute to the EclipseLink log. The statistics contains three sets of information:

- **Info**; statistics that are constant informational data, such as the session name, or time of login.
- **Counter**; statistics that are cumulative counters of total operations, such as cache hits, or query executions.
- **Timer**; statistics that are cumulative measurements of total time (in nano seconds) for a specific type of operation, reading, writing, database operations.

Statistics are generally grouped in total and also by query type, query class, and query name. Counters and timers are generally recorded for the same operations, so the time per operation could also be calculated.

The time between statistic dumps can be configured by using the `setDumpTime(long)` method in the `PerformanceMonitor` class. If dumping the results is not desired, then the `dumpTime` attribute can be set to be very large such as `Long.MAX_VALUE`. The statistic can also be accessed in a Java program with the `getOperationTime(String)` method.

The performance monitor can also be configured with a profile weight. The profile weights are defined in the `SessionProfiler` class and used by the PerformanceMonitor class. The weights include:

- `NONE`—No statistics are recorded.
- `NORMAL`—Informational statistics are recorded.
- `HEAVY`—Informational, counter and timer statistics are recorded.
- `ALL`—All statistics are recorded (this is the default).

```
+
```

In the current release, the performance monitor responds with the same information for the `HEAVY` and `ALL` values.

Task 3: Measure Fetch Group Field Usage

Use the Fetch Group Monitor to measure fetch group field usage. This can be useful for performance analysis in a complex system.

Enable this monitor by using the system property `org.eclipse.persistence.fetchgroupmonitor=true`.

The monitor outputs the attribute used for a class every time a new attribute is accessed.

Task 4: 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 5: Modify Poorly-Performing Application Components provides some guidelines for dealing with problems in each of these areas.

Task 5: Modify Poorly-Performing Application Components

For each source of application performance problems listed in Task 4: Identify Sources of Application Performance Problems, you can try specific workarounds, as described in this section.

Identifying General Performance Optimizations

Avoid overriding EclipseLink 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. These defaults may include:

- Batch writing – See "jdbc.batch-writing" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.
- Statement caching – See "jdbc.cache-statements" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.
- Read and write connection pool size – See "connection-pool" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.
- Session cache size – See "maintain-cache" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Use the Workbench rather than manual coding. These tools are not only easy to use: the default configuration they export to deployment XML (and the code it generates, if required) represents best practices optimized for most applications.

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, as 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.

Possible ways to optimize the schema include:

- Aggregating two tables into one
- Splitting one table into many
- Using a collapsed hierarchy
- Choosing one out of many

See "Data Storage Schema" in *EclipseLink Concepts* for additional information.

Mappings and Descriptors

If you find performance bottlenecks in your mapping and descriptors, try these solutions:

- Always use indirection (lazy loading). It is not only critical in optimizing database access, but also allows EclipseLink to make several other optimizations including optimizing its cache access and unit of work processing. See "cache-usage" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.
- Avoid using method access in your EclipseLink 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 "Using Method or Direct Field Access" in the *EclipseLink Concepts*.
- Avoid using the existence checking option checkCacheThenDatabase on descriptors, unless required by the application. The default existence checking behavior offers better performance. See "@ExistenceChecking" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.
- Avoid expensive initialization in the default constructor that EclipseLink uses to instantiate objects. Instead, use lazy initialization or use an EclipseLink instantiation policy to configure the descriptor to use a different constructor. See "@InstantiationCopyPolicy" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Cache

You can often improve performance through caching, even in a clustered environment 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 "Understanding Caching" in *EclipseLink Concepts* and the following examples:

- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/Caching`
- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/CacheCoordination`
- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/DCN`

Data Access

Depending on the type of data source your application accesses, EclipseLink offers a variety of `Login` options that you can use to tune the performance of low level data reads and writes. For optimizing higher-level data reads and writes, "Understanding Data Access" in *EclipseLink Concepts* 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.

Queries

EclipseLink provides an extensive query API for reading, writing, and updating data. "Understanding EclipseLink Queries" in *EclipseLink Concepts* 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.

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 EclipseLink application and persistence.
- For your application or Jakarta EE server, ensure 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.

Task 6: Measure Performance Again

Finally, after identifying possible performance bottlenecks and taking some action on them, rerun your application, again with the profiler enabled (see Enabling the EclipseLink Profiler). Review the results and, if more action is required, follow the procedures outlined in Task 5: Modify Poorly-Performing Application Components.

Chapter 19. Exposing JPA Entities Through RESTful Data Services

This chapter describes how to expose JPA persistence units using RESTful Data services.

This chapter includes the following sections:

Use Case

Expose persistent data model and application logic over REST for the development of Thin Server Architecture (TSA) clients including HTML5/JavaScript and mobile technologies.

Solution

Use RESTful Data Services to expose entities using a RESTful service, without writing JAX-RS code.

Components

- A Jakarta EE application server with the following:
  - EclipseLink 2.4 or later.
  - Support for Java API for RESTful Web Services (JAX-RS) 1.0, for example the JAX-RS reference implementation, Jersey (see `http://jersey.java.net/`).
- A compliant Java Database Connectivity (JDBC) database, such as Oracle Database, Oracle Express, or MySQL

19.1. Introduction to the Solution

REpresentational State Transfer (REST) defines a set of architectural principles for distributed systems, in which Web Services are viewed as resources. Those resources are identified by URIs and can be addressed and transferred using the HTTP protocol. REST can be used with a number of technologies, including JPA. HTTP methods are used to access and perform operations on resources.

The Java API for RESTful Web Services (JAX-RS) is an API designed to make it easy to develop Java applications that use the REST architecture. With JAX-RS, you use annotations to define resources and the actions that can be performed on those resources.

While it is possible to use JAX-RS directly to interact with JPA persistence units in a RESTful application, RESTful Data Services provide an API that makes it easier to implement REST for JPA persistence,. You can use this API to interact with JPA persistence units without explicitly writing JAX-RS code, thus providing a simple way to expose persistence units through REST.

For an example that uses JAX-RS directly to implement JPA persistence in a RESTful application, see "RESTful Service Example" at http://wiki.eclipse.org/EclipseLink/Examples/REST/GettingStarted. For information about simplifying that process by using RESTful Data Services, continue reading this chapter.

RESTful Data Services are made available via a web fragment, which extends the capabilities of a web application. The REST functionality is made available by including the RESTful Data Services JAR file in the `WEB-INF/lib` folder of a web application.

The RESTful Data Services runtime provides access to all persistence units packaged in the application in which it is running, as well as any dynamic persistence units that are provisioned within it.

19.2. Implementing the Solution

This section contains the following tasks for exposing JPA entities using RESTful Data Services:

- Step 1: Prerequisites
- Step 2: Create and Configure the Application
- Step 3: Understand RESTful Data Services URI Basics
- Step 4: Represent Entities Using JPA, JAXB, or JSON
- Step 5: Issue Client Calls for Operations on the Persistence Unit
- Step 6: Implement Security
- Step 7: Understand the Structure of RESTful Data Services Responses

Step 1: Prerequisites

To implement and use RESTful Data Services, you need:

- Either of the following Jakarta EE application servers:
  - Oracle WebLogic Server Release 4.0 or later.
  - Glassfish Server 3.1.2 or later.

With Glassfish Server 3.1.2, you must upgrade the EclipseLink version to use the version of the RESTful Data Services shipped in EclipseLink 2.4.2 (and must also include DBWS). See http://www.eclipse.org/eclipselink/downloads/ for EclipseLink downloads.

Those servers include the following: **EclipseLink 2.4 or later, configured as the persistence provider.** Jersey, the reference implementation of the Java API for RESTful Web Services (JAX-RS) 1.0 specification. * The org.eclipse.persistence.jpars_``version_num`.jar` file, where `*version_num*` is the version of the jpars file, for example, org.eclipse.persistence.jpars_2.4.1.v20121003-ad44345.jar. This file is included in the EclipseLink distributions from the Eclipse foundation, at http://www.eclipse.org/eclipselink/downloads/: **In the installer distribution, the file is located in `eclipselink\jlib\jpa\`.** In the bundles

distribution, the file is located with the other bundles. * Any compliant Java Database Connectivity (JDBC) database, including Oracle Database, Oracle Database Express Edition (Oracle Database XE), or MySQL. These instructions are based on Oracle Database XE 11g Release 2. For the certification matrix, see

Step 2: Create and Configure the Application

RESTful Data Services are designed to function with standard JPA applications, with little extra work required beyond enabling the service, as described below:

1. Develop an application using one or more standard JPA persistence units, package it in a Web ARchive (WAR) file, and deploy it normally.

The fragment must be placed inside a WAR, because it offers Web services. That WAR may optionally be packaged inside an Enterprise Archive (EAR) file.

Weaving is required for several RESTful Data Services features to work: providing relationships as links, editing relationships, and dealing with lazy many-to-one relationships. Therefore, for those features, you must either deploy to a Jakarta EE compliant server or statically weave your classes.

2. Include the RESTful Data Services servlet in the WAR containing the application. (For instructions on downloading, see.Step 1: Prerequisites)

The RESTful Data Services JAR file includes a `web-fragment.xml` file that identifies the servlet and defines the root URI for the RESTful service.

Add the `org.eclipse.persistence.jpars_``version_num`.jar` file to the WAR containing the application, under `WEB-INF/lib`.

Step 3: Understand RESTful Data Services URI Basics

URIs used for making REST calls for RESTful Data Services follow these standard patterns:

- The base URI for an application is: `http://server:port/application-name/persistence/{version}`

As of EclipseLink 2.4.2, support for using RESTful Data Services URIs without a version number is deprecated and will be removed in future releases. The version of RESTful Data Services in EclipseLink 2.4.2 is `v1.0`, and that version number should be used to make REST requests to RESTful Data Services.

- For base operations on the persistence unit, add the persistence unit name: `/persistence/{version}/{unit-name}`
- For specific types of operations, add the type of operation, for example:
  - Entity operations: `/persistence/{version}/{unit-name}/entity`
  - Query operations: `/persistence/{version}/{unit-name}/query`

- Single result query operations: `/persistence/{version}/{unit-name}/singleResultQuery`
- Persistence unit level metadata operations: `/persistence/{version}/{unit-name}/metadata`
- Base operations: `/persistence/{version}`

For complete documentation on how to construct these URIs, see RESTful Data Services API Reference.

Step 4: Represent Entities Using JPA, JAXB, or JSON

Entities in RESTful Data Services are represented in two ways:

- **As JPA Entities** - The mappings of the JPA entities must be represented in the typical JPA fashion, using either annotations or XML files. These mappings are used to interact with the data source.
- **As JAXB/JSON** - No specific mapping information is required when using JAXB/JSON. By default, RESTful Data Services use the JAXB defaults (defined in the JAXB specification) to map to JAXB/JSON. You can optionally provide JAXB annotations on the classes to alter the way the objects are mapped. Additionally, the persistence unit property `eclipselink.jpa-rs.oxm` can be specified in a persistence unit's `persistence.xml` to specify XML-defined JAXB mappings.

Relationships

In general, JAXB default mappings are sufficient to allow information exchange using JSON/JAXB. There are, however, some special cases when dealing with relationships.

Bidirectional Relationships and Cycles

Bidirectional relationships are typical in JPA and are easy to represent in a database using foreign keys. They are more difficult to represent in an XML or JSON document using standard JAXB. However, the EclipseLink JAXB implementation provides a way to define an inverse relationship. Inverse relationships are not directly written to XML or JSON but are populated when the XML or JSON is unmarshalled. The way this is handled is as follows:

JPA bidirectional relationships are defined to have an owning side and a non-owning side. The entity that has the table with a foreign key in the database is the *owning* entity. The other table—the one pointed to—is the *inverse* (non-owning) entity. JPA mapping provides a mapped-by attribute that defines which is which. The `mappedBy` attribute must be on the inverse side. RESTful Data Services default the owning side to be an inverse relationship. As a result, when an object with an owned relationship is read or written, that relationship is ignored.

Consider the following pseudo-code:

```
@Entity
ClassA{

@Id
int id
```

```
  @OneToOne
  myB

}


  @Entity
  ClassB{

  @Id
  int id

  @OneToOne(mappedby="myB")
  myA

}
```

If the objects are identified as follows...

- `A1` with `id=1` and `myB = B1`
- `B1` with `id=11` and `myA = A1`

...the following JSON corresponds to those objects:

```
  A {
    id:1
  }

  B {
    id:11
    myA: {
      id: 1
    }
  }
```

Passing By Value vs. Passing By Reference

RESTful Data Services allow relationship objects to be passed either by value or by reference in the REST request. JSON attributes hold resource references (see "Pass By Value"), while `_relationship`s have "navigation" links (see "Pass By Reference").

Pass By Value

To pass an object by value, create typical JSON or XML that represents the object. The following JSON passes `myA` by value:

```
  B {
    id:11
```

```
    myA {
      id: 1
    }
  }
```

Pass By Reference

To pass an object by reference, use a `_link`. The link represents the RESTful Data Services call necessary to get that object. The following JSON passes `myA` by reference:

```
  B {
    id:11
    myA {
      _link:{
          href: "http://localhost:8080/app/persistence/v1.0/pu/entity/A/1"
          method: "GET"
          rel: "self"
      }
    }
  }
```

A `link` consists of `href`, `method` and `rel` attributes.

- The `href` (Hypertext REFerence) is the URI of the entity linked to. The `href` uniquely identifies the linked entity or attribute.
- The `method` identifies the operation the `href` is to be used for.
- The `rel` represents the relationship between the containing entity and the entity linked to.

Lists can mix and match items represented by reference and by value. The corresponding entity must exist if an item is represented by reference in a request; otherwise RESTful Data Services returns an error.

The following example shows JSON that can be sent to RESTful Data Services as a request, in a regular-expression-like syntax:

```
{
    "numericAttribute": 1
    "stringAttribute": "auction1"
    "dateAttribute": 12-09-16
    "singleRelatedItem": RELATED_ITEM?
      "listRelatedItem":
      {
          RELATED_ITEM*
      }
  }


  RELATED_ITEM =
```

```
      {
        "numericAttribute": 11
        "stringAttribute": "myName"
      }

  OR

      "_link" {
        "rel"="self",
        "href" = "LINK_HREF",
        "method"="GET"
     }
```

The following JSON represents an entity called `Auction` with several directly mapped fields and a collection of an entity called `Bid`.

```
  {
    "description": "Auction 1",
    "endPrice": 0,
    "id": 2,
    "image": "auction1.jpg",
    "name": "A1",
    "sold": false,
    "startPrice": 100,
    "bids": [
      {
        "_link": {
       "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/auction/entity/Bid/5",
        "method": "GET",
        "rel": "self"
      }
    },
    {
      "_link": {
        "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/auction/entity/Bid/6",
        "method": "GET",
        "rel": "self"
      }
    }
  ]
  }
```

XML representation mimics the JSON representation. The following is sample XML for an entity called `Auction`, with several directly mapped attributes and a list of an entity called `Bid`.

```
<?xml version="1.0" encoding="UTF-8"?>
<Auction>
    <description>Auction 1</description>
    <endPrice>0.0</endPrice>
    <id>2</id>
    <image>auction1.jpg</image>
    <name>A1</name>
    <sold>false</sold>
    <startPrice>100.0</startPrice>
    <bids>
      <_link
href="http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/auction/entity/Bid
/5" method="GET" rel="self" />
    </bids>
    <bids>
      <_link
href="http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/auction/entity/Bid
/6" method="GET" rel="self" />
    </bids>
  </Auction>
```

Step 5: Issue Client Calls for Operations on the Persistence Unit

Clients use HTTP calls to perform operations on persistence units in a deployed application. The requirements and options for constructing the calls are described in RESTful Data Services API Reference.

Specify Media Format in the Header

This REST interface can handle both XML and JSON representations of data. The caller is responsible for using HTTP header values to indicate the format of the content:

- `Content-Type = application/json` indicates that the content being sent is JSON
- `Content-Type = application/xml` indicates that the content being sent is XML
- `Accept = application/json` indicates that the expected format of the result is JSON
- `Accept = application/xml` indicates that the expected format of the result is XML

If no header value is specified, JSON is used by default. If `Content-type` is specified and `Accept` is not specified, the returned format matches the `Content-type` passed in.

In many REST utilities, the `Accept` value is defaulted to `application/xml`. In those cases, you must configure this value explicitly if you want JSON.

About Logging

Messages related to RESTful Data Services operations are logged to a logger called `org.eclipse.persistence.jpars`. Most messages are logged at the `FINE` level. Exception stacks are

logged at `FINER`.

Messages related to operations within `EntityManager`s, `EntityManagerFactory`s and `JAXBContext`s are logged in the same manner as other EclipseLink logging.

Step 6: Implement Security

Secure RESTful Data Services through typical REST security mechanisms.

Step 7: Understand the Structure of RESTful Data Services Responses

The RESTful Data Services response messages, either in XML or in JSON, contain following categories:

- Basic data types, such as `int`, `double`, `String`, `Integer`, `Double`, `Boolean`, etc.
- Relationships (links and relationships)

The next sections explain the semantic and syntactic details of each category of data.

There is also a minor generic difference between the XML and JSON responses (other than format). The JSON responses do not include the root name of an entity, while XML responses do. See the `employee` root/grouping name in the XML response below. The root name is derived from the name of the entity it represents.

JSON

```
{
   "firstName":"John",
   "lastName": "Smith",
   ...
}
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<employee>
   <firstName>John</firstName>
   <lastName>Smith</lastName>
   ...
</employee>
```

Basic Data Types

In the RESTful Data Services responses, basic data types and primitives are presented as simple JSON or XML fields. For example:

JSON

```
{
   "firstName":"John",
   "lastName": "Smith",
   ...
}
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<employee>
   <firstName>John</firstName>
   <lastName>Smith</lastName>
   ...
</employee>
```

Links and Relationships

RESTful Data Services operations return all relationships by reference, with the exception of JPA embeddables and element collections.

The `relationships` are links pointing to the (JPA) relationships of an entity, such as one-to-one and one-to-many. For example, assume that an employee has multiple phone numbers (one-to-many). When the employee is read, the response will contain a relationship link pointing to the relationship between the employee and the phone entities, plus a list of the links, with each link pointing to a (unique) phone number that the employee owns. For example:

```
{
   "firstName": "Jacob",
   "gender": "Male",
   "id": 743627,
   "lastName": "Smith",
   "version": 1,
   "_relationships": [
      {
         "_link": {
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/hr/entity/Employee/743627/ph
oneNumbers",
            "rel": "phoneNumbers"
         }
      }
   ],

   "phoneNumbers": [
      {
         "_link": {
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/hr/entity/PhoneNumber/743627
```

```
+cell",
            "method": "GET",
            "rel": "self"
         }
      },
      {
         "_link": {
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/hr/entity/PhoneNumber/743627
+work",
            "method": "GET",
            "rel": "self"
         }
      }
   ]
}
```

Embedded objects and element collections are strictly privately-owned (dependent) objects.They have no identity, and there is no cascade option on an `ElementCollection`. The target objects are always persisted, merged, and removed with their parent. Therefore, RESTful Data Services embeds these objects directly in responses, rather than providing links to them. For example, assume the `Employee` object has `EmploymentPeriod` defined as `Embedded`. When the `Employee` is read, the response will contain `EmploymentPeriod` as an embedded object, not a link to it. Relationships are currently not supported for embedded attributes. See the example below:

```
{
   "firstName": "John",
   "lastName": "Smith",

   "employmentPeriod": {
      "startDate": "2010-04-23T14:12:03.905-04:00",
      "endDate": "2013-01-23T12:00:02.301-04:00",
      "_relationships": []
   },
   ...
}
```

Similarly, element collections are also directly contained in RESTful Data Services responses as collections, not as links. For example, assume the `Employee` object has a `"certifications"` attribute defined as a collection of `Certification` objects. When the `Employee` is read, the response will contain list of `Certification` objects, not links:

```
{
   "firstName": "John",
   "lastName": "Smith",
   "certifications": [
      {
         "issueDate": "2013-04-23T15:02:23.071-04:00",
```

```
            "name": "Java"
         },
         {
            "issueDate": "2010-05-23T11:02:23.033-04:00",
            "name": "Weblogic"
         }
      ],
      ...
   }
```

19.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- "Building RESTful Web Services with JAX-RS" in The Jakarta EE 6 Tutorial at http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html.
- "RESTful Service Example" at http://wiki.eclipse.org/EclipseLink/Examples/REST/GettingStarted.
- JSR 311: JAX-RS: The Java API for RESTful Web Services" at http://jcp.org/en/jsr/detail?id=311
- Jersey project at http://jersey.java.net/.

19.4. RESTful Data Services API Reference

The following types of RESTful operations can be used with JPA via HTTP when using RESTful Data Services:

- Entity Operations
- Entity Operations on Relationships
- Query Operations
- Base Operations
- Metadata Operations

19.5. Entity Operations

Entity operations are those performed against a specific entity type within the persistence unit.

The base URI for entity operations is as follows:

`/persistence/{version}/{unit-name}/entity/{type}/*`

The `{type}` value refers to the type name (descriptor alias).

Supported entity operations are:

- FIND
- PERSIST
- MERGE
- DELETE

FIND

HTTP Request Syntax

`GET /persistence/{version}/{unit-name}/entity/{type}/{id}?{hints}`

where:

- `{id}` is a string
- `` `hints` `` are specified using HTTP query parameters, with the key being the name of the EclipseLink query hint

Example

`GET http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo/1`

Produces

JSON or XML

Response

- `OK`, with a payload containing the entity
- `NOT_FOUND` if the entity does not exist

Usage

Composite Keys

Composite keys are supported. The `` ` character is reserved and therefore cannot be used in fields that represent keys. Composite keys are separated using the ` `` character and should be specified in an order corresponding to the Java default sorting of the attribute names.

For example, consider an entity `Phone`, with attributes `extB=123` and `extA=321`. The URL to find the entity is:

`http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Phone/321+123`

The `321` comes before the `123` because `extA` comes before `extB` when sorted in Java.

Result Caching

Default EclipseLink and HTTP caching is enabled and configured through standard means.

Refresh

The `EntityManager.refresh` operation can be invoked using the `find` with the query hint for `Refresh`.

Attributes

Navigating into the attributes of an entity (for example, to get the `Address` entity associated with an employee in a single REST request) is supported to one level, for example:

`/persistence/v1.0/{unit-name}/entity/{type}/{id}/{relationship}` will work

while

`/persistence/v1.0/{unit-name}/entity/{type}/{id}/{relationship}/{index}/{relationship2}` will not

PERSIST

HTTP Request Syntax

`PUT /persistence/{version}/{unit-name}/entity/{type}`

Example

`PUT http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo`

Consumes

JSON or XML

Payload

Entity

Produces

JSON or XML

Response

Payload containing the entity returned by the persist operation

Usage

`PUT` is required to be idempotent. As a result, it will fail if called with an object that expects the server to provide an ID field. Typically this will occur if the metadata specifies a generated key and the field that contains that key is unpopulated.

MERGE

HTTP Request Syntax

`POST /persistence/{version}/{unit-name}/entity/{type}`

Example

`POST http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo`

Consumes

JSON or XML

Payload

Entity

Produces

JSON or XML

Response

Payload containing the entity returned by the merge operation.

Merge takes an object graph and makes it part of the persistence context through comparison. It compares the object and all related objects to the ones that already exist and issues `INSERT`s, `UPDATE`s, and `DELETE`s to put the object in the persistence context.

DELETE

HTTP Request Syntax

`DELETE /persistence/{version}/{unit-name}/entity/{type}{id}`

where `{id}` is defined using a string

Example

`DELETE http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo/1`

Response

`OK`

19.6. Entity Operations on Relationships

The base URI for relationship operations is as follows:

`/persistence/{version}/{unit-name}/entity/{entity}/{id}/{relationship}`

Supported relationship operations are:

- READ

- ADD
- REMOVE

READ

Use this operation to get the values of a relationship.

HTTP Request Syntax

`GET /persistence/{version}/{unit-name}/entity/{type}/{id}/{relationship}`

where:

- `{id}` is a string.
- `{relationship}` is the JPA name of the relationship.

Example

`GET http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo/1/myRelationship`

Produces

JSON or XML

Response

- `OK`, Payload containing an entity or a list of entities.
- `NOT_FOUND` if the entity does not exist

ADD

Use this operation to add to a list or replace the value of a many-to-one relationship.

HTTP Request Syntax

`POST /persistence/{version}/{unit-name}/entity/{type}/{id}/{relationship}?{partner}`

As of EclipseLink 2.4.2, `partner` should be specified as a query parameter. Specifying `partner` as a matrix parameter is deprecated.

Examples

For unidirectional relationships, `{partner}` is not required, for example:

`POST http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo/1/myRelationship`

For bi-directional relationships, you must provide the name of the attribute that makes up the opposite side of the relationship. For example, to update an `Auction.bid` where the opposite side of

the relationship is `Bid.auction`, use the following:

```
POST   http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo/1/myRelationship?partner=bid
```

Consumes

JSON or XML

Payload

Entity with the new value.

Relationship objects can be passed by value or by reference. See "Passing By Value vs. Passing By Reference".

Produces

JSON or XML

Response

Payload containing the entity with the added element

REMOVE

Use this operation to remove a specific entity from the list or a null on a many-to-one relationship.

HTTP Request Syntax

```
DELETE                                                    /persistence/{version}/{unit-name}/entity/{type}/{id}/{relationship}?{relationshipListItemId}
```

where `relationshipListItemId` is an optional query parameter. The `relationshipListItemId` is meaningful only when the `{relationship}` to be removed is a list. The `relationshipListItemId` should be set to the `id` of a member in the relationship list when only that member of the relationship list needs to be removed. The entire list specified by the `{relationship}` will be removed when `relationshipListItemId` is not specified.

Example

```
DELETE http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/entity/Foo/1/myRelationship
```

Consumes

JSON or XML

Relationship objects can be passed by value or by reference. See "Passing By Value vs. Passing By Reference".

Produces

JSON or XML

Response

- OK
- Payload containing the entity with the removed element

19.7. Query Operations

The base URI for query operations is as follows:

```
GET /persistence/{version}/{unit-name}/query/{name}{params}
```

The following query operations are supported:

Named queries doing reads can be run two ways in JPA. Both are supported in the REST API. They are:

- Query Returning List of Results
- Update/Delete Query

Query Returning List of Results

HTTP Request Syntax

```
GET /persistence/{version}/{unit-name}/query/{name};{parameters}? {hints}
```

where:

- `*parameters*` are specified using HTTP matrix parameters
- `*hints*` are specified using HTTP query parameters and with the key being the name of the EclipseLink query hint

Examples

```
GET http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/query/Foo.findByName;name=myname
```

```
GET http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/query/Foo.findByName;name=myname?eclipselink.jdbc.max-results=500
```

Produces

JSON or XML

Response

A payload containing a list of entities. An XML response contains a `List` as a grouping name for a

collection of items and `item` as a grouping name for each member of a collection returned. JSON responses use square brackets `[]` to encapsulate a collection and curly braces `{}` to encapsulate each member of a collection. For example:

XML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<List>
   <item>
      <firstName>Miles</firstName>
      <lastName>Davis</lastName>
      <manager>
         <firstName>Charlie</firstName>
         <lastName>Parker</lastName>
         <gender>Male</gender>
         <id>26</id>
      </manager>
   </item>
   <item>
      <firstName>Charlie</firstName>
      <lastName>Parker</lastName>
      <manager>
         <firstName>Louis</firstName>
         <lastName>Armstrong</lastName>
         <gender>Male</gender>
         <id>27</id>
      </manager>
   </item>
</List>
```

JSON Example

```
[
   {
      "firstName": "Miles",
      "lastName": "Davis",
      "manager": {
         "firstName": "Charlie",
         "lastName": "Parker",
         "gender": "Male",
         "id": 26
      }
   },
   {
      "firstName": "Charlie",
      "lastName": "Parker",
      "manager": {
         "firstName": "Louis",
         "lastName": "Armstrong",
```

```
        "gender": "Male",
        "id": 27
      }
   }
]
```

Update/Delete Query

HTTP Request Syntax

```
POST /persistence/{version}/{unit-name}/query/{name};parameters?hints
```

where:

- `*parameters*` are specified using HTTP matrix parameters
- `*hints*` are specified using HTTP query parameters and with the key being the name of the EclipseLink query hint

Examples

```
POST                        http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/query/Foo.deleteAllByName;name=myname
```

```
POST                        http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/query/Foo.updateName;name=myname?eclipselink.jdbc.max-results=500
```

Produces

JSON or XML

Response

A payload containing the number of entities updated or deleted

---

19.8. Single Result Queries

HTTP Request Syntax

```
GET /persistence/{version}/{unit-name}/singleResultQuery/{name};{parameters}?{hints}
```

where:

- `*parameters*` are specified using HTTP matrix parameters
- `*hints*` are specified using HTTP query parameters and with the key being the name of the EclipseLink query hint

Example

```
GET
http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/singleResultQuery/Foo.findByName;name=myname
```

Produces

JSON, XML, or application/octet-stream

Response

A payload containing an entity

19.9. Base Operations

Base operations are:

- List Existing Persistence Units

List Existing Persistence Units

HTTP Request Syntax

```
GET /persistence/{version}
```

Example

```
GET http://localhost:8080/exampleApp/persistence/v1.0
```

Produces

JSON or XML

Response

A payload containing a list of persistence unit names and links to metadata about them. For example:

```
[
   {

      "_link": {
         "href":
"http://localhost:8080/exampleApp/persistence/v1.0/employee/metadata",
         "method": "application/json",
         "rel": "employee"
      }
   },
   {
      "_link": {
```

```
        "href":
"http://localhost:8080/exampleApp/persistence/v1.0/traveler/metadata",
        "method": "application/json",
        "rel": "traveler"
      }
   }
]
```

19.10. Metadata Operations

The following metadata operations are supported:

- List Types in a Persistence Unit
- List Queries in a Persistence Unit
- Describe a Specific Entity

List Types in a Persistence Unit

HTTP Request Syntax

`GET /persistence/{version}/{unit-name}/metadata`

Example

`GET http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/metadata`

Produces

JSON

Response

- `OK`, with a payload containing a list of types, with links to more detailed metadata, for example:

```
{
    "persistenceUnitName": "hr",
    "types": [
        {
            "_link": {
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/metadata/entity/E
mployee",
                "method": "application/json",
                "rel": "Employee"
            }
        },
```

```
        {
            "_link": {
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/metadata/entity/P
honeNumber",
                "method": "application/json",
                "rel": "PhoneNumber"
            }
        }
    ]
}
```

- `NOT_FOUND` if the persistence unit is not found

List Queries in a Persistence Unit

HTTP Request Syntax

`GET /persistence/{version}/{unit-name}/metadata/query`

Example

`GET http://localhost:8080/exampleApp/persistence/v1.0/ExamplePU/metadata/query`

Produces

JSON

Response

- `OK` with a payload containing a list of all available queries, for example:

```
[
    {
        "queryName": "Employee.count",
        "returnTypes": [
            "Long"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.co
unt",
            "rel": "execute"
        },
        "jpql": "SELECT count(e) FROM Employee e"
    },
    {
        "queryName": "EmployeeAddress.getRegion",
```

```
        "returnTypes": [
            "String",
            "String",
            "String"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/EmployeeAdd
ress.getRegion",
            "rel": "execute"
        },
        "jpql": "SELECT u.postalCode, u.province, u.street FROM EmployeeAddress u"
    },
    {
        "queryName": "Employee.getPhoneNumbers",
        "returnTypes": [
            "String",
            "String",
            "PhoneNumber"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.ge
tPhoneNumbers",
            "rel": "execute"
        },
        "jpql": "SELECT e.firstName, e.lastName, pn FROM Employee e JOIN
e.phoneNumbers pn"
    },
    {
        "queryName": "EmployeeAddress.getPicture",
        "returnTypes": [
            "byte[]"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/EmployeeAdd
ress.getPicture;id={id}",
            "rel": "execute"
        },
        "jpql": "SELECT u.areaPicture FROM EmployeeAddress u where u.id = :id"
    },
    {
        "queryName": "EmployeeAddress.updatePostalCode",
        "returnTypes": [
            "EmployeeAddress"
        ],
        "linkTemplate": {
```

```
            "method": "post",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/EmployeeAdd
ress.updatePostalCode;postalCode={postalCode};id={id}",
            "rel": "execute"
        },
        "jpql": "UPDATE EmployeeAddress u SET u.postalCode = :postalCode where u.id
= :id"
    },
    {
        "queryName": "Employee.salaryMax",
        "returnTypes": [
            "int",
            "Object"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.sa
laryMax",
            "rel": "execute"
        },
        "jpql": "SELECT e.id, max(e.salary) AS max_salary from Employee e GROUP BY
e.id, e.salary"
    },
    {
        "queryName": "EmployeeAddress.getAll",
        "returnTypes": [
            "EmployeeAddress"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/EmployeeAdd
ress.getAll",
            "rel": "execute"
        },
        "jpql": "SELECT u FROM EmployeeAddress u"
    },
    {
        "queryName": "EmployeeAddress.getById",
        "returnTypes": [
            "EmployeeAddress"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/EmployeeAdd
ress.getById;id={id}",
            "rel": "execute"
        },
```

```
        "jpql": "SELECT u FROM EmployeeAddress u where u.id = :id"
    },
    {
        "queryName": "Employee.getManagerById",
        "returnTypes": [
            "String",
            "String",
            "Employee"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.ge
tManagerById;id={id}",
            "rel": "execute"
        },
        "jpql": "select u.firstName, u.lastName, u.manager from Employee u where
u.id = :id"
    },
    {
        "queryName": "Employee.findAll",
        "returnTypes": [
            "Employee"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.fi
ndAll",
            "rel": "execute"
        },
        "jpql": "SELECT e FROM Employee e ORDER BY e.id"
    },
    {
        "queryName": "Employee.getManager",
        "returnTypes": [
            "String",
            "String",
            "Employee"
        ],
        "linkTemplate": {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.ge
tManager",
            "rel": "execute"
        },
        "jpql": "select u.firstName, u.lastName, u.manager from Employee u"
    }
]
```

- `NOT_FOUND` if persistence unit is not found

Describe a Specific Entity

HTTP Request Syntax

`GET /persistence/{version}/{unit-name}/metadata/entity/` `type`

Example

`GET http://localhost:8080/CustomerApp/persistence/v1.0/Inventory/metadata/entity/Customer`

Produces

JSON

Response

- `OK`, with a payload containing details about the entity and available operations on it, for example,


```
{
    "name": "Employee",
    "attributes": [
        {
            "name": "id",
            "type": "int"
        },
        {
            "name": "firstName",
            "type": "String"
        },
        {
            "name": "gender",
            "type": "Gender"
        },
        {
            "name": "lastName",
            "type": "String"
        },
        {
            "name": "salary",
            "type": "double"
        },
        {
            "name": "version",
            "type": "Long"
        },
        {
            "name": "period",
```

```
            "type": "EmploymentPeriod"
        },
        {
            "name": "manager",
            "type": "Employee"
        },
        {
            "name": "office",
            "type": "Office"
        },
        {
            "name": "address",
            "type": "EmployeeAddress"
        },
        {
            "name": "certifications",
            "type": "List<Certification>"
        },
        {
            "name": "responsibilities",
            "type": "List<String>"
        },
        {
            "name": "projects",
            "type": "List<Project>"
        },
        {
            "name": "expertiseAreas",
            "type": "List<Expertise>"
        },
        {
            "name": "managedEmployees",
            "type": "List<Employee>"
        },
        {
            "name": "phoneNumbers",
            "type": "List<PhoneNumber>"
        }
    ],
    "linkTemplates": [
        {
            "method": "get",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/entity/Employee/{
primaryKey}",
            "rel": "find"
        },
        {
            "method": "put",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/entity/Employee",
```

```
            "rel": "persist"
        },
        {
            "method": "post",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/entity/Employee",
            "rel": "update"
        },
        {
            "method": "delete",
            "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/entity/Employee/{
primaryKey}",
            "rel": "delete"
        }
    ],
    "queries": [
        {
            "queryName": "Employee.count",
            "returnTypes": [
                "Long"
            ],
            "linkTemplate": {
                "method": "get",
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.co
unt",
                "rel": "execute"
            },
            "jpql": "SELECT count(e) FROM Employee e"
        },
        {
            "queryName": "Employee.getPhoneNumbers",
            "returnTypes": [
                "String",
                "String",
                "PhoneNumber"
            ],
            "linkTemplate": {
                "method": "get",
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.ge
tPhoneNumbers",
                "rel": "execute"
            },
            "jpql": "SELECT e.firstName, e.lastName, pn FROM Employee e JOIN
e.phoneNumbers pn"
        },
        {
            "queryName": "Employee.salaryMax",
            "returnTypes": [
```

```
                "int",
                "Object"
            ],
            "linkTemplate": {
                "method": "get",
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.sa
laryMax",
                "rel": "execute"
            },
            "jpql": "SELECT e.id, max(e.salary) AS max_salary from Employee e GROUP
BY e.id, e.salary"
        },
        {
            "queryName": "Employee.getManagerById",
            "returnTypes": [
                "String",
                "String",
                "Employee"
            ],
            "linkTemplate": {
                "method": "get",
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.ge
tManagerById;id={id}",
                "rel": "execute"
            },
            "jpql": "select u.firstName, u.lastName, u.manager from Employee u
where u.id = :id"
        },
        {
            "queryName": "Employee.findAll",
            "returnTypes": [
                "Employee"
            ],
            "linkTemplate": {
                "method": "get",
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.fi
ndAll",
                "rel": "execute"
            },
            "jpql": "SELECT e FROM Employee e ORDER BY e.id"
        },
        {
            "queryName": "Employee.getManager",
            "returnTypes": [
                "String",
                "String",
                "Employee"
            ],
```

```
            "linkTemplate": {
                "method": "get",
                "href":
"http://localhost:8080/eclipselink.jpars.test/persistence/v1.0/hr/query/Employee.ge
tManager",
                "rel": "execute"
            },
            "jpql": "select u.firstName, u.lastName, u.manager from Employee u"
        }
    ]
}
```

- `NOT_FOUND` if the persistence unit is not found

Chapter 20. Using Database Events to Invalidate the Cache

This chapter describes EclipseLink Database Change Notification (DCN), which allows you to use caching with a shared database in JPA.

This chapter includes the following sections:

Use Case

Users want to use a shared cache with their JPA application, however, external applications update the same database data, or the cache is in a clustered environment. The cache may retain stale data.

Solution

EclipseLink provides an API which allows the database to notify EclipseLink of database changes. The changed objects can then be invalidated in the EclipseLink shared cache. Stale data can be discarded, even if other applications access the same data in the database.

Components

- Oracle 11*g*R2 (11.2) (or higher) database
- EclipseLink 2.4 or later.
  - EclipseLink library: eclipselink.jar
  - JDBC library: `ojdbc6.jar`.
  - JPA library: `persistence.jar`.

Sample

For sample files that illustrate the use of Database Change Notification and shared caching in an application that uses JPA, see "EclipseLink/Examples/JPA/DCN" in the EclipseLink documentation.

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

20.1. Introduction to the Solution

EclipseLink provides a shared (L2) object cache that can reduce database access for objects and their relationships. This cache is enabled by default which is normally not a problem, unless the contents of the database are modified directly by other applications, or by the same application on other servers in a clustered environment. This may result in stale data in the cache.

If the contents of the database are modified, then a mechanism is needed to ensure that the contents of the cache are synchronized with the database. That mechanism is provided by EclipseLink Database Change Notification. DCN allows shared caching to be used in the JPA environment.

Database Change Notification extends the functionality provided by the Oracle Database Continuous Query Notification feature. For more information, see "Continuous Query Notification" in *Oracle Database JDBC Developer's Guide*.

EclipseLink Database Change Notification extends the functionality provided by the Oracle Database Continuous Query Notification. One of the features of Continuous Query Notification is that it allows database events to be raised when rows in a table are modified.

To detect modifications, EclipseLink DCN uses the `ROWID` to inform of row level changes in the primary table. EclipseLink includes the `ROWID` in all queries for a DCN-enabled class. EclipseLink also selects the object's `ROWID` after an insert operation. EclipseLink maintains a cache index on the `ROWID`, in addition to the object's `Id`. EclipseLink also selects the database transaction ID once for each transaction to avoid invalidating the cache on the server that is processing the transaction.

EclipseLink DCN is enabled through the `OracleChangeNotificationListener` (`org.eclipse.persistence.platform.database.oracle.dcn.OracleChangeNotificationListener`) listener class. This listener integrates with Oracle JDBC to receive database change events. To enable the listener, specify the full path to the `OracleChangeNotificationListener` class as the value of the `eclipselink.cache.database-event-listener` property in the `persistence.xml` file.

By default, all entities in the domain are registered for change notification. However, you can selectively disable change notification for certain classes by tagging them in the Java files with the `databaseChangeNotificationType` (`org.eclipse.persistence.annotations.DatabaseChangeNotificationType`) attribute of the `Cache` annotation. The value of this attribute determines the type of database change notification an entity should use. The default value of the `databaseChangeNotificationType` attribute is `Invalidate`. To disable change notification for a class, set the value of the attribute to `None`.

The `databaseChangeNotificationType` attribute is relevant only if the persistence unit has been configured with a database event listener, such as the `OracleChangeNotificationListener` class, that receives database change events. This allows the EclipseLink cache to be invalidated or updated from database changes.

Oracle strongly suggests that you use optimistic locking (writes on stale data will fail and automatically invalidate the cache) in your transactions. If you include an `@Version` annotation in your entity, then the version column in the primary table will always be updated, and the object will always be invalidated.

20.2. Implementing the Solution

This section contains the following tasks to enable shared caching in a JPA environment:

- Task 1: Set up the Database and Tables

Task 1: Set up the Database and Tables

The solution presumes that you are working with an Oracle 11gR2 (11.2) or higher database that contains the tables that you are interested in.

Task 2: Grant User Permissions

Among other permissions, the database user must be granted the `CHANGE NOTIFICATION` privilege. To do this, you must have a DBA privilege, such as `SYS`, or have your database administrator apply it:

`grant change notification to` *`user`*

The following example illustrates granting the change notification privilege to user `SCOTT`.

```
...
define user="SCOTT"
define pass="tiger"
grant create session, alter session to &&user
/
grant resource, connect to &&user
/
grant select any dictionary to &&user
/
grant select any table to &&user
/
grant change notification to &&user
/
...
```

Task 3: Set the Classpath

Ensure that the `eclipselink.jar` EclipseLink library, the `ojdbc6.jar` JDBC library, the `persistence.jar` JPA library, and the domain classes are present on the classpath.

Task 4: Identify Classes that will Participate in Change Notification

By default, all entities in the domain will participate in change notification. There are several different ways to limit the entities that will participate. For example, the entity classes can be indicated by the `<entity class ···>` element in the `orm.xml` file, indicated with the `<exclude-unlisted-classes>` element in the `persistence.xml` file, or contained in a JAR file.

The `<exclude-unlisted-classes>` element is not intended for use in the Java SE environment.

Entity classes can also be excluded by using a `Cache` annotation attribute in the Java files. For more information, see Exclude Classes from Change Notification (Optional).

Another way to identify the entity classes is to use the `<class>` element in the `persistence.xml` file. The following example indicates that the `Order`, `OrderLine`, and `Customer` classes in the `model` package will participate in change notification. For an example of a complete `persistence.xml` file, see Example 20-1.

```
...
<class>model.Order</class>
<class>model.OrderLine</class>
<class>model.Customer</class>
...
```

Task 5: Add the Database Event Listener

Use the `eclipselink.cache.database-event-listener` property to identify the database event listener. The `org.eclipse.persistence.platform.database.oracle.dcn.OracleChangeNotificationListener` class is the listener for EclipseLink Database Change Notification. This allows the EclipseLink cache to be invalidated by database events.

The following example illustrates the `eclipselink.cache.database-event-listener` property configured with the `OracleChangeNotificationListener` class. For an example of a complete `persistence.xml` file, see Example 20-1.

```
...
   <properties>
      <property name="eclipselink.cache.database-event-listener"
value="org.eclipse.persistence.platform.database.oracle.dcn.OracleChangeNotificationLi
stener"/>
   </properties>
...
```

Note that you can also use:

```
<property name="eclipselink.cache.database-event-listener" value="DCN">
```

Example 20-1 illustrates an example of a complete `persistence.xml` file. The classes that will participate in change notification are the `Order`, `OrderLine`, and `Customer` classes from the `model` package. The `eclipselink.cache.database-event-listener` property is set to the full path of the `OracleChangeNotificationListener` class.

A `<provider>` tag is optional if running in a container where EclipseLink is the

> default provider.

Example 20-1 Sample persistence.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<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_2_0.xsd"
                version="2.0">
    <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>model.Order</class>
        <class>model.OrderLine</class>
        <class>model.Customer</class>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.cache.database-event-listener" value="DCN"/>
        </properties>
    </persistence-unit>
</persistence>
```

Task 6: Edit the Java Files

Typically, to participate in change notification, no changes are needed to the Java classes which correspond to database tables. However, setting optimistic locking with the `@Version` annotation is strongly suggested.

If you want to exclude classes that are listed in the persistence unit, you can tag them in the Java files. EclipseLink tracks changes only to the primary table. If you want changes to secondary tables to also be tracked, you can indicate this in the Java files.

Set Optimistic Locking

Oracle strongly suggests that you use optimistic locking: writes on stale data will fail and automatically invalidate the cache. Include an `@Version` annotation in your entity; the version column in the primary table will always be updated, and the older version of the object will always be invalidated.

In Example 20-2 the `@Version` annotation is defined for the entity `Customer`. Note that getters and setters are defined for the `version` variable.

Example 20-2 Defining the @Version Annotation

```
...
@Entity
@Table(name="DBE_CUSTOMER")
public class Customer implements Serializable {
    @Id
```

```
    @GeneratedValue(generator="CUST_SEQ")
    @TableGenerator(name="CUST_SEQ")
    @Column(name="CUST_NUMBER")
    private long id;

 @Version
    private long version;

 ...
    public long getVersion() {
        return version;
    }

    public void setVersion(long version) {
        this.version = version;
    }
...
```

Exclude Classes from Change Notification (Optional)

Use the `databaseChangeNotificationType` attribute of the `Cache` annotation to identify the classes for which you do not want change notifications. To exclude a class from change notification, set the attribute to `DatabaseChangeNotificationType.NONE`, as illustrated in the following example.

```
...
@Entity
@Cache(databaseChangeNotificationType=DatabaseChangeNotificationType.NONE)
public class Order {
...
```

Track Changes in Secondary Tables (Optional)

EclipseLink tracks changes only to the primary table. If any updates occur in a secondary table, EclipseLink will not invalidate the object. If you want changes to secondary tables to be tracked as well, add the `@Version` annotation to the entity.

Oracle DCN listens only for events from the primary table. It does not track changes in secondary tables, or relationships tables. The reason for this is that Oracle DCN only tracks the `ROWID`, so there is no correlation from the `ROWID` of the primary, secondary and relationship tables. Thus, to receive events when a secondary or relationship table changes, the version in the primary table must change so that the event is returned.

20.3. Limitations on the Solution

EclipseLink Database Change Notification has the following limitations:

- Changes to an object's secondary tables will not trigger it to be invalidate unless a `@Version` annotation is used and updated in the primary table.

- Changes to an object's `OneToMany`, `ManyToMany`, and `ElementCollection` relationships will not trigger it to be invalidate, unless an `@Version` annotation is used and updated in the primary table.

20.4. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- "Continuous Query Notification" in *Oracle Database JDBC Developer's Guide*.
- "EclipseLink/Examples/JPA/DCN"—This page describes an example of cache sharing in a JPA environment that employs DCN. Sample files and instructions for running the example are included. See the EclipseLink documentation:

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

Chapter 21. Using EclipseLink with NoSQL Databases

This chapter describes how Oracle TopLink supports the ability to map objects to NoSQL database systems such as internet databases, object databases, XML databases, and even legacy databases.

This chapter includes the following sections:

Use Case

Users need to use EclipseLink with NoSQL data sources.

Solution

EclipseLink provides support for multiple NoSQL data sources. This solution illustrates using Oracle NoSQL and MongoDB.

Components

- EclipseLink 2.4 or later
- NoSQL datasource.
- JCA Adapter.

Sample

See the following EclipseLink samples for related information:

- `http://wiki.eclipse.org/EclipseLink/Examples/JPA/NoSQL`

21.1. Introduction to the Solution

EclipseLink supports access to NoSQL data through the JavaEE Connector Architecture. You must use a JCA adapter (provided by EclipseLink, a third party, or custom built).

Most NoSQL data is hierarchical in form so using embeddable objects is common. Some NoSQL adaptors support XML data, so NoSQL mapped objects can use XML mappings when mapping to XML.

21.2. Implementing the Solution

This section contains the following tasks for converting objects to and from JSON documents.

Task 1: Prerequisites

Task 2: Mapping the Data

You can configure mappings to NoSQL data with the EclipseLink `@NoSQL` annotation and `<no-sql>` XML element. The `@NoSQL` annotation defines the class as mapping to non-relational data. You can use `@NoSQL` with JPA Entity or Embeddable classes.

The `@NoSQL` annotation allows you to specify the `dataType` and `dataFormat` of the data. The `dataType` will vary, depending on your NoSQL datasource:

- For MongoDB, `dataType` is the collection name that the JSON documents are stored to.
- For Oracle NoSQL, `dataType` is the first part of the major key value.

The `dataFormat` depends on the type structure (data format) of data being stored.

- For MongoDB, use `MAPPED` for its structured database.
- For Oracle NoSQL, use `MAPPED` (for key/value data) or `XML` (for a single XML document).

Example 21-1 illustrates how to use `@NoSQL` with `@Entity` and `@Embeddable` classes.

Example 21-1 Using @NoSql Annotation with JSON

```
@Entity
@NoSQL(dataType="orders", dataFormat=DataFormatType.MAPPED)
public class Order {
  @Id
  @GeneratedValue
  @Field(name="_id")
  private long id;
  @Basic
  @Field(name="description")
  private String description;
  @Embedded
  @Field(name="deliveryAddress")
  private Address deliveryAddress
  @ElementCollection
  @Field(name="orderLines")
  private List<OrderLine> orderLines;
  @ManyToOne
```

```
  @JoinField(name="customerId")
  private Customer customer;
}

@Embeddable
@NoSQL(dataFormat=DataFormatType.MAPPED)
public class OrderLine {
    @Field(name="lineNumber")
    private int lineNumber;
    @Field(name="itemName")
    private String itemName;
    @Field(name="quantity")
    private int quantity;
}
```

Task 3: Defining IDs

With EclipseLink, you can use any field (or set of fields) as your ID when using a non-relational database, just like any other relational Entity. You can use a *natural* ID (that is, assigned by the application) or a *generated* ID (that is, assigned by EclipseLink).

MongoDB also requires an `_id` field in every document. If no `_id` field is present, Mongo will automatically generate and assign the `_id` field using an OID (object identifier), which is similar to a UUID (universally unique identifier).

- To use a natural ID as the Mongo ID, simply name the field as `_id` by using the `@Field` (or `@Column`) annotation without any of the relational details.

  For example:

  ```
  @Field(name="_id")
  private long id;
  ```

- To use the generated Mongo OID as your ID, simply include `@Id`, `@GeneratedValue`, and `@Field(name="_id")` annotations in the object's ID field mapping.

  The `@GeneratedValue` tells EclipseLink to use the Mongo OID to generate this ID value. To use a UUID instead of the Mongo OID, use the @UUIDGenerator annotation.

MongoDB does not support `@SequenceGenerator` or `@TableGenerator` nor the `IDENTITY`, `TABLE`, and `SEQUENCE` generation types.

The ID of the Mongo OID or UUID is not a numerical value; you must map it as a `String` or `byte[]`.

For example:

```
@Id
@GeneratedValue
```

```
@Field(name="_id")
private String id;
```

Task 4: Defining Mappings

With non-relational databases, EclipseLink maps objects to structured data such as XML or JSON. NoSQL supports all existing JPA mapping annotations and XML, including embedded data and embedded collections. If you do not define a mapping annotation (or XML) for an attribute EclipseLink uses the default mapping.

Basic Mappings

Because the NoSQL defaults follow the JPA defaults, most simple mappings do not require any configuration. Field names used in the Mongo BSON document will mirror the object attribute names (in uppercase). To use a different BSON field name, use the `@Field` annotation.

Do not use `@Column` or `@JoinColumn`. Instead use `@Field` and `@JoinField`. Additionally, the `@JoinTable` and `@CollectionTable` annotations are not supported or required.

Embedded Values

Use the `@Embedded` annotation to persist embedded values and the `@ElementCollection` annotation for embedded collections. Because all data is stored in the XML document, no separate table (that is, `@CollectionTable`) is needed. Additionally, because embedded objects are nested in the document and do not require unique field names, the `@AttributeOverride` attribute is not needed.

Normally, you will not need to use the `@Embedded` annotation, since it will default correctly.

However, EclipseLink does not default `@ElementCollection` mappings, therefore you must include that annotation.

Relationships

You should use the relationship annotations (such as `@OneToOne`, `@ManyToOne`, `@OneToMany` and `@ManyToMany`) only with *external* relationships. Relationships *within* the document should use the Embedded Values.

EclipseLink fully supports external relationships to other documents by using a foreign key. The ID of the target object is stored in the source object's document. For a collection, a collection of IDs is stored. Use the `@JoinField` annotation to define the name of the foreign key field in the BSON document.

EclipseLink does not support the `mappedBy` option for relationships with non-relational databases, as the foreign keys would need to be stored on both sides.

You can also define a relationship mapping by using a query. However you must use a `DescriptorCustomizer` instead of an annotation.

Example 21-2 Sample Mappings

```
@Basic
private String description;
@Basic
private double totalCost = 0;
@Embedded
private Address billingAddress;
@Embedded
private Address shippingAddress;
@ElementCollection
private List<OrderLine> orderLines = new ArrayList<OrderLine>();
@ManyToOne(fetch=FetchType.LAZY)
private Customer customer;
```

Task 5: Using Locking

Locking support is dependent on the NoSQL platform. Some NoSQL platforms may offer support for optimistic version locking.

- Oracle NoSQL – Locking is not supported.
- MongoDB – Version locking is supported.

MongoDB does not support transactions. If a lock error occurs during a transaction, any objects that have been previously written will not be rolled back.

If the NoSQL platform does not support locking, you can use the `@Version` annotation (as shown in Example 21-3) to validate objects on `merge()` operations.

Example 21-3 Using @Version

```
@Version
private long version;
...
```

Task 6: Defining Queries

Querying in NoSQL is dependent on the NoSQL platform. Some NoSQL data-sources may support dynamic querying through their own query language, others may not support querying at all.

JPQL Queries

The Java Persistence Query Language (JPQL) is the query language defined by JPA. JPQL can be used for reading (`SELECT`), as well as bulk updates (`UPDATE`) and deletes (`DELETE`). You can use JPQL in a `NamedQuery` (through annotations or XML) or in dynamic queries using the EntityManager `createQuery()` API.

- Oracle NoSQL – Supports `find()` and JPQL and Criteria by Id or with no WHERE clause.
- MongoDB – Supports JPQL and Criteria queries, with some restrictions: joins, sub-selects, group by and certain database functions are not supported.

Example 21-4 Oracle NoSQL JPQL Examples

```
```

Example 21-5 MongoDB JPQL Examples

```
Query query = em.createQuery("Select o from Order o where o.totalCost > 1000");
List<Order> orders = query.getResultList();
```

```
Query query = em.createQuery("Select o from Order o where o.description like
'Pinball%'");
List<Order> orders = query.getResultList();
```

```
Query query = em.createQuery("Select o from Order o join o.orderLines l where
l.description = :desc");
query.setParameter("desc", "shipping");
List<Order> orders = query.getResultList();
```

```
Query query = em.createQuery("Select o.totalCost from Order o");
List<BigDecimal> orders = query.getResultList();
```

Native Queries

Native SQL queries are not translated, and passed directly to the database. SQL queries can be used for advanced queries that require database specific syntax.

Although native SQL queries are not supported with NoSQL, some NoSQL platforms have their own, native query language. EclipseLink supports JPA native queries using that language.

- MongoDB – Supports JPA native queries by using the MongoDB native command language.

Example 21-6 Oracle NoSQL Native Query

```
```

Example 21-7 MongoDB Native Query

```
Query query = em.createNativeQuery("db.ORDER.findOne({\"_id\":\"" + oid + "\"})",
Order.class);
```

```
Order order = (Order)query.getSingleResult();
```

Task 7: Connecting to the Database

EclipseLink connects to NoSQL databases through the persistence.xml file. Use the `<eclipselink.target-database>` property to define the specific NoSQL platform. You must also define a connection with the `<eclipselink.nosql.connection-spec>` property. Additional connection values (such as the `db`, `port`, and `host` can also be defined.

To connect to a cluster of Mongo databases, enter a comma, separated list of values for the `host` and `port`.

Example 21-8 Oracle NoSQL persistence.xml Example

```
```

Example 21-9 MongoDB persistence.xml Example

```
<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_2_0.xsd"
version="2.0">
    <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.target-database"
value="org.eclipse.persistence.nosql.adapters.mongo.MongoPlatform"/>
            <property name="eclipselink.nosql.connection-spec"
value="org.eclipse.persistence.nosql.adapters.mongo.MongoConnectionSpec"/>
            <property name="eclipselink.nosql.property.mongo.port" value="27017,
27017"/>
            <property name="eclipselink.nosql.property.mongo.host" value="host1,
host2"/>
            <property name="eclipselink.nosql.property.mongo.db" value="acme"/>
        </properties>
    </persistence-unit>
</persistence>
```

21.3. Additional Resources

See the following resources for more information about the technologies and tools used to implement the solutions in this chapter:

- *Developing JAXB Applications Using EclipseLink MOXy*
- *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*

- EclipseLink Extensions Incubator: http://wiki.eclipse.org/EclipseLink/Development/Incubator/Platform

Chapter 22. Using EclipseLink with the Oracle Database

This chapter provides instructions for understanding and usingEclipseLink features that are designed specifically to support the Oracle Database platform.

This chapter includes the following sections:

- Introduction to the Solution
- Implementing the Solution
- Additional Resources

Use Case

EclipseLink offers a persistence solution that is designed to work with any database. However, applications that plan to use the Oracle Database platform can take advantage of enhanced support for the Oracle Database.

Solution

The solution is achieved by using various EclipseLink APIs and Oracle products. Applications choose to implement different EclipseLink APIs based on the Oracle Database feature or products being used.

Components

- EclipseLink 2.4 or later.
- Oracle Database
- Additional Oracle Database and Middleware products are required depending on the features that an application chooses to use.

22.1. Introduction to the Solution

EclipseLink includes enhanced support for the Oracle Database platform. Applications that have standardized on the Oracle Database can take advantage of this support to gain ease-of-use, increase performance and scalability, and enhance security. EclipseLink includes support for native Oracle JDBC-specific APIs, PL/SQL, Oracle Real Application Clusters (RAC), Oracle Virtual Private Database, Oracle Proxy Authentication, and Oracle Spatial and Graph. Refer to the Oracle Database documentation for details on these technologies.

Much of the Oracle Database platform support is contained in the `org.eclipse.persistence.platform.database.oracle*` package. For details on the APIs, see *Java API Reference for EclipseLink*.

22.2. Implementing the Solution

The solution in this section is organized according to technology. The organization allows developers to easily understand the different parts of the solution and choose specific parts to implement.

This section includes the following topics:

- Using Oracle Platform-Specific APIs
- Using Oracle PL/SQL With EclipseLink
- Using Oracle Virtual Private Database
- Using Oracle Proxy Authentication
- Using EclipseLink with Oracle RAC
- Using Oracle Spatial and Graph

Using Oracle Platform-Specific APIs

Oracle Database platform support is provided in the `org.eclipse.persistence.platform.database.OraclePlatform` class, the `org.eclipse.persistence.platform.database.oracle*` packages, and the `org.eclipse.persistence.mappings.xdb` package for Oracle XML Database support. For details on the API, see *Java API Reference for EclipseLink*. For details on specific Oracle SQL types, see *Oracle Database JDBC Java API Reference*.

The following support is provided for the Oracle Database:

- Batch writing with optimistic locking
- Native SQL for `byte[]`, `Date`, `Time`, `Timestamp` and `Calendar`
- Support for `BLOB` and `CLOB` database types using Oracle JDBC specific `LOBLocator` for large values

For non-Oracle thin JDBC drivers or applications environments where the thin driver is wrapped, it is possible to turn off `LOBLocator` usage using `setShouldUseLocatorForLOBWrite(boolean)` on the platform instance.

- Native support for outer join syntax `(+) =`
- Native Sequencing (`SELECT SEQ_NAME.NEXTVAL FROM DUAL`)
- Native SQL/ROWNUM support for `MaxRows` and `FirstResult` filtering.
- Hierarchical selects (connect by prior)
- Returning clause
- Custom expression functions (`REGEXP_LIKE`, `LOCATE`, `ATAN2`, `LOG`, `CONCAT`, `SYSDATE` (Date, Time, Today), `EXCEPT`)
- PLSQL data types, stored functions, stored procedure syntax for invoking and parameter passing, output parameters and output cursors. See Using Oracle PL/SQL With EclipseLink.

- Timestamp query for use in optimistic locking using `SYSDATE` and `SYSTIMESTAMP`
- Multi-byte support of `NCHAR`, `NSTRING`, and `NCLOB`
- Support of `TIMESTAMP`, `TIMESTAMPTZ`, and `TIMESTAMPLTZ`
- Oracle XML Database support of `XMLType` field and custom XSQL functions (`extract`, `extractValue`, `existsNode`, `isFragment`, `getStringVal`, and `getNumberVal`)
- XDK XML parser
- Flashback Querying in Historical Sessions
- Object-relational Mappings (ReferenceMapping, StructureMapping, NestedTableMapping, ArrayMapping, ObjectArrayMapping)
- Oracle AQ
- Oracle Real Application Clusters. See Using EclipseLink with Oracle RAC.
- Virtual Private Database (VPD), including Oracle Label Security. Using Oracle Virtual Private Database.
- Proxy Authentication. See Using Oracle Proxy Authentication.

Using Oracle PL/SQL With EclipseLink

EclipseLink includes APIs for use with Oracle PL/SQL. The APIs are located in the `org.eclipse.persistence.platform.database.oracle.plsql` package and the `org.eclipse.persistence.platform.database.oracle.annotations` package.

This Section contains the following topics:

- Executing an Oracle PL/SQL Stored Function
- Handling PL/SQL arguments for Oracle Stored Procedures

Executing an Oracle PL/SQL Stored Function

Oracle PL/SQL stored functions can be used to return complex PL/SQL data-types such as `RECORD` types and `TABLE` types. PL/SQL types are not supported by Oracle JDBC, so these types must be translated to Oracle `OBJECT` types and `VARRAY` types. `OBJECT` types are returned as `java.sql.Struct` and `VARRAY` as `java.sql.Array` types in JDBC.

Executing PL/SQL stored functions or procedures requires defining mirror `OBJECT` and `VARRAY` types for the `RECORD` and `TABLE` types. `OBJECT` types can be mapped to classes annotated with either `@Entity` or `@Embeddable` using the `@Struct` annotation. Typically, classes annotated with `@Embeddable` are used, unless the `OBJECT` type defines an `Id` and can be stored in a table. Nested `OBJECT` and `VARRAY` types are mapped using the `@Structure` and `@Array` annotations.

Use the `PLSQLStoredFunctionCall` class or the `@NamedPLSQLStoredFunctionQuery` annotation to call a stored function using PL/SQL types. The `PLSQLStoredProcedureCall` class and the `@NamedPLSQLStoredProcedureQuery` annotation also exist for stored procedures. Use the `StoredFunctionCall` class, the `@NamedStoredFunctionQuery` annotation, the `StoredProcedureCall` class, and the `@NamedStoredProcedureQuery` annotation for stored functions and procedure that do not return complex PL/SQL types.

Main Tasks

To execute an Oracle PL/SQL stored function:

- Task 1: Create an Oracle Stored Function That Returns a PL/SQL Record Type
- Task 2: Define an Object Type Mirror
- Task 3: Define a Java Class Mapping The OBJECT Type
- Task 4: Execute a PL/SQL Stored Function Using JpaEntityManager
- Task 5: Define a Stored Function Using @NamedPLSQLStoredFunctionQuery
- Task 6: Use the Stored Function in a Query

Task 1: Create an Oracle Stored Function That Returns a PL/SQL Record Type

```
CREATE OR REPLACE PACKAGE EMP_PKG AS
TYPE EMP_REC IS RECORD (F_NAME VARCHAR2(30), L_NAME VARCHAR2(30),
   SALARY NUMBER(10,2));
FUNCTION GET_EMP RETURN EMP_REC;
END EMP_PKG;

CREATE OR REPLACE PACKAGE BODY EMP_PKG AS
FUNCTION GET_EMP RETURN EMP_REC AS
   P_EMP EMP_REC;
   BEGIN P_EMP.F_NAME := 'Bob'; P_EMP.F_NAME := 'Smith'; P_EMP.SALARY := 30000;
   RETURN P_EMP;
END;
END EMP_PKG;
```

Task 2: Define an Object Type Mirror

```
CREATE OR REPLACE TYPE EMP_TYPE AS OBJECT (F_NAME VARCHAR2(30),
   L_NAME VARCHAR2(30), SALARY NUMBER(10,2))
```

Task 3: Define a Java Class Mapping The OBJECT Type

```
@Embeddable
@Struct(name="EMP_TYPE", fields={"F_NAME", "L_NAME", "SALARY"})
public class Employee {
   @Column(name="F_NAME")
   private String firstName;
   @Column(name="L_NAME")
   private String lastName;
   @Column(name="SALARY")
   private BigDecimal salary;
   ...
}
```

Task 4: Execute a PL/SQL Stored Function Using JpaEntityManager

```
import jakarta.persistence.Query;
import org.eclipse.persistence.platform.database.orcle.plsql.
   PLSQLStoredFunctionCall;
import org.eclipse.persistence.queries.ReadAllQuery;

DataReadQuery databaseQuery = new DataReadQuery();
databaseQuery.setResultType(DataReadQuery.VALUE);
PLSQLrecord record = new PLSQLrecord();
record.setTypeName("EMP_PKG.EMP_REC");
record.setCompatibleType("EMP_TYPE");
record.setJavaType(Employee.class);
record.addField("F_NAME", JDBCTypes.VARCHAR_TYPE, 30);
record.addField("L_NAME", JDBCTypes.VARCHAR_TYPE, 30);
record.addField("SALARY", JDBCTypes.NUMERIC_TYPE, 10, 2);
PLSQLStoredFunctionCall call = new PLSQLStoredFunctionCall(record);
call.setProcedureName("EMP_PKG.GET_EMP");
databaseQuery.setCall(call);

Query query = ((JpaEntityManager)entityManager.getDelegate()).
   createQuery(databaseQuery);
Employee result = (Employee)query.getSingleResult();
```

Task 5: Define a Stored Function Using @NamedPLSQLStoredFunctionQuery

```
@NamedPLSQLStoredFunctionQuery(name="getEmployee", functionName="EMP_PKG.GET_EMP",
   returnParameter=@PLSQLParameter(name="RESULT", databaseType="EMP_PKG.EMP_REC"))
@Embeddable
@Struct(name="EMP_TYPE", fields={"F_NAME", "L_NAME", "SALARY"})
@PLSQLRecord(name="EMP_PKG.EMP_REC", compatibleType="EMP_TYPE",
   javaType=Employee.class,fields={@PLSQLParameter(name="F_NAME"),
@PLSQLParameter(name="L_NAME"), @PLSQLParameter(name="SALARY",
   databaseType="NUMERIC_TYPE")})

public class Employee {
 ...
}
```

Task 6: Use the Stored Function in a Query

```
Query query = entityManager.createNamedQuery("getEmployee");
Employee result = (Employee)query.getSingleResult();
```

Handling PL/SQL arguments for Oracle Stored Procedures

The standard way of handling a stored procedure is to build an instance of the `StoredProcedureCall` class. However, the arguments must be compatible with the JDBC specification. To handle Oracle

PL/SQL arguments (for example, `BOOLEAN`, `PLS_INTEGER`, PL/SQL record, and so on), use the `PLSQLStoredProcedureCall` class.

the `PLSQLStoredProcedureCall` class is only supported on Oracle8 or higher.

Using the PLSQLStoredProcedureCall Class

The following example demonstrates handling PL/SQL arguments using the `PLSQLStoredProcedureCall` class. The example is based on the following target procedure:

```
PROCEDURE bool_in_test(x IN BOOLEAN)
```

Example of Using the PLSQLStoredProcedureCall Class

```
import java.util.List;
import java.util.ArrayList;
import org.eclipse.persistence.logging.SessionLog;
import org.eclipse.persistence.platform.database.jdbc.JDBCTypes;
import org.eclipse.persistence.platform.database.oracle.Oracle10Platform;
import org.eclipse.persistence.platform.database.oracle.OraclePLSQLTypes;
import org.eclipse.persistence.platform.database.oracle.PLSQLStoredProcedureCall;
import org.eclipse.persistence.queries.DataModifyQuery;
import org.eclipse.persistence.sessions.DatabaseLogin;
import org.eclipse.persistence.sessions.DatabaseSession;
import org.eclipse.persistence.sessions.Project;
import org.eclipse.persistence.sessions.Session;

public class TestClass {

   public static String DATABASE_USERNAME = "username";
   public static String DATABASE_PASSWORD = "password";
   public static String DATABASE_URL = "jdbc:oracle:thin:@localhost:1521:ORCL";
   public static String DATABASE_DRIVER = "oracle.jdbc.driver.OracleDriver";

   public static void main(String[] args) {
      Project project = new Project();
      DatabaseLogin login = new DatabaseLogin();
      login.setUserName(DATABASE_USERNAME);
      login.setPassword(DATABASE_PASSWORD);
      login.setConnectionString(DATABASE_URL);
      login.setDriverClassName(DATABASE_DRIVER);
      login.setDatasourcePlatform(new Oracle10Platform());
      project.setDatasourceLogin(login);
      Session s = project.createDatabaseSession();
      s.setLogLevel(SessionLog.FINE);
      ((DatabaseSession)s).login();

      PLSQLStoredProcedureCall call = new PLSQLStoredProcedureCall();
      call.setProcedureName("bool_in_test");
```

```
      call.addNamedArgument("X", OraclePLSQLTypes.PLSQLBoolean);
      DataModifyQuery query = new DataModifyQuery();
      query.addArgument("X");
      query.setCall(call);
      List queryArgs = new ArrayList();
      queryArgs.add(Integer.valueOf(1));
      s.executeQuery(query, queryArgs);
    }
}
```

The following log excerpt shows the target procedure being invoked from an anonymous PL/SQL block:

```
...[EclipseLink Info]: 2007.11.23 01:03:23.890--DatabaseSessionImpl(15674464)--
   Thread(Thread[main,5,main])-- login successful
[EclipseLink Fine]: 2007.11.23 01:03:23.968--DatabaseSessionImpl(15674464)--
   Connection(5807702)--Thread(Thread[main,5,main])--
DECLARE
   X_TARGET BOOLEAN := SYS.SQLJUTL.INT2BOOL(:1);
BEGIN
   bool_in_test(X=>X_TARGET);
END;
   bind => [:1 => 1]
```

Notice the conversion of the Integer to a PL/SQL `BOOLEAN` type in the `DECLARE` stanza (as a similar conversion is used for OUT `BOOLEAN` arguments).

Mixing JDBC Arguments With Non JDBC Arguments

A Stored Procedure may have a mix of regular and non JDBC arguments. Use the `PLSQLStoredProcedureCall` class when at least one argument is a non JDBC type. In addition, some additional information may be required for the JDBC type (length, scale or precision) because the target procedure is invoked from an anonymous PL/SQL block. The example is based on the following target procedure:

```
PROCEDURE two_arg_test(x IN BOOLEAN, y IN VARCHAR)
```

Example of Mixing JDBC Arguments With NonJDBC Arguments

```
import org.eclipse.persistence.platform.database.jdbc.JDBCTypes;
...
   PLSQLStoredProcedureCall call = new PLSQLStoredProcedureCall();
   call.setProcedureName("two_arg_test");
   call.addNamedArgument("X", OraclePLSQLTypes.PLSQLBoolean);
   call.addNamedArgument("Y", JDBCTypes.VARCHAR_TYPE, 40);
   DataModifyQuery query = new DataModifyQuery();
   query.addArgument("X");
```

```
   query.addArgument("Y");
   query.setCall(call);
   List queryArgs = new ArrayList();
   queryArgs.add(Integer.valueOf(0));
   queryArgs.add("test");
   boolean worked = false;
   String msg = null;
   s.executeQuery(query, queryArgs);
```

The following log excerpt shows the target procedure being invoked from an anonymous PL/SQL block:

```
[EclipseLink Fine]: 2007.11.23 02:54:46.109--DatabaseSessionImpl(15674464)--
   Connection(5807702)--Thread(Thread[main,5,main])--
DECLARE
   X_TARGET BOOLEAN := SYS.SQLJUTL.INT2BOOL(:1);
   Y_TARGET VARCHAR(40) := :2;
BEGIN
   two_arg_test(X=>X_TARGET, Y=>Y_TARGET);
END;
   bind => [:1 => 0, :2 => test]
```

Handling IN and OUT Arguments

The following example demonstrates a stored procedure that contain both `IN` and `OUT` arguments and is based on the following target procedure:

```
PROCEDURE two_arg_in_out(x OUT BINARY_INTEGER, y IN VARCHAR) AS
BEGIN
   x := 33;
END;
```

Example of Handling IN and OUT Arguments

```
import static org.eclipse.persistence.platform.database.oracle.OraclePLSQLTypes.
   BinaryInteger;
...
   PLSQLStoredProcedureCall call = new PLSQLStoredProcedureCall();
   call.setProcedureName("two_arg_in_out");
   call.addNamedOutputArgument("X", OraclePLSQLTypes.BinaryInteger);
   call.addNamedArgument("Y", JDBCTypes.VARCHAR_TYPE, 40);
   DataReadQuery query = new DataReadQuery();
   query.setCall(call);
   query.addArgument("Y");
   List queryArgs = new ArrayList();
   queryArgs.add("testsdfsdfasdfsdfsdfsdfsdfsdfdfsdfsdffds");
   boolean worked = false;
```

```
   String msg = null;
   List results = (List)s.executeQuery(query, queryArgs);
   DatabaseRecord record = (DatabaseRecord)results.get(0);
   BigDecimal x = (BigDecimal)record.get("X");
   if (x.intValue() != 33) {
      System.out.println("wrong x value");
   }
```

The following log excerpt shows the target procedure being invoked from an anonymous PL/SQL block:

```
[EclipseLink Fine]: 2007.11.23 03:15:25.234--DatabaseSessionImpl(15674464)--
   Connection(5807702)--Thread(Thread[main,5,main])--
DECLARE
   Y_TARGET VARCHAR(40) := :1;
   X_TARGET BINARY_INTEGER;
BEGIN
   two_arg_in_out(X=>X_TARGET, Y=>Y_TARGET);
   :2 := X_TARGET;
END;
   bind => [:1 => testsdfsdfasdfsdfsdfsdfsdfsdfdfsdfsdffds, X => :2]
```

The order in which arguments are bound at runtime must be altered. Anonymous PL/SQL blocks must process the ordinal markers (`:1`,`:2`) for all the IN arguments first, then the OUT arguments. Inside the block, the arguments are passed in the correct order for the target procedure, but the bind order is managed in the `DECLARE` stanza and after the target procedure has been invoked.

Handling IN OUT Arguments

Anonymous PL/SQL blocks cannot natively handle `IN OUT` arguments. The arguments must be split into two parts: an IN-half and an OUT-half. The following example demonstrates a stored procedure that handles IN OUT arguments and is based on the following target procedure:

```
PROCEDURE two_args_inout(x VARCHAR, y IN OUT BOOLEAN) AS
BEGIN
   y := FALSE;
END;
```

Example of Handling IN OUT Arguments

```
...
   PLSQLStoredProcedureCall call = new PLSQLStoredProcedureCall();
   call.setProcedureName("two_args_inout");
   call.addNamedArgument("X", JDBCTypes.VARCHAR_TYPE, 20);
   call.addNamedInOutputArgument("Y", OraclePLSQLTypes.PLSQLBoolean);
   DataReadQuery query = new DataReadQuery();
```

```
   query.addArgument("X");
   query.addArgument("Y");
   query.setCall(call);
   List queryArgs = new ArrayList();
   queryArgs.add("test");
   queryArgs.add(Integer.valueOf(1));
   List results = (List)s.executeQuery(query, queryArgs);
   DatabaseRecord record = (DatabaseRecord)results.get(0);
   Integer bool2int = (Integer)record.get("Y");
   if (bool2int.intValue() != 0) {
      System.out.println("wrong bool2int value");
   }
```

The following log excerpt shows the target procedure being invoked from an anonymous PL/SQL block:

```
[EclipseLink Fine]: 2007.11.23 03:39:55.000--DatabaseSessionImpl(25921812)--
   Connection(33078541)--Thread(Thread[main,5,main])--
DECLARE
   X_TARGET VARCHAR(20) := :1;
   Y_TARGET BOOLEAN := SYS.SQLJUTL.INT2BOOL(:2);
BEGIN
   two_args_inout(X=>X_TARGET, Y=>Y_TARGET);
   :3 := SYS.SQLJUTL.BOOL2INT(Y_TARGET);
END;
   bind => [:1 => test, :2 => 1, Y => :3]
```

The `Y` argument is split in two using the `:2` and `:3` ordinal markers.

Using Oracle Virtual Private Database

EclipseLink supports Oracle Virtual Private Database (VPD). Oracle VPD is a server-enforced, fine-grained access control mechanism. Oracle VPD ties a security policy to a table by dynamically appending SQL statements with a predicate to limit data access at the row level. You can create your own security policies, or use Oracle's custom implementation called Oracle Label Security (OLS). For details about Oracle VPD, see *Oracle Database Security Guide*. For details about Oracle Label Security, see *Oracle Label Security Administrator's Guide*.

For details about using Oracle VPD with Multitenancy, see Using VPD Multi-Tenancy.

To use the Oracle Database VPD feature in an EclipseLink application, an isolated cache should be used. Any entity that maps to a table that uses Oracle VPD should have the descriptor configured as isolated. In addition, you typically use exclusive connections.

To support Oracle VPD, you must implement session event handlers that the are invoked during the persistence context's life cycle. The session event handler you must implement depends on whether or not you are using Oracle Database proxy authentication.

Oracle VPD with Oracle Database Proxy Authentication

By using Oracle Database proxy authentication, you can set up Oracle VPD support entirely in the database. That is, rather than session event handlers to execute SQL, the database performs the required setup in an after login trigger using the proxy session_user.

For details on using Oracle proxy authentication, see Using Oracle Proxy Authentication.

Oracle VPD Without Oracle Database Proxy Authentication

If you are not using Oracle Database proxy authentication, implement session event handlers for the following session events:

- `postAcquireExclusiveConnection`: used to perform Oracle VPD setup at the time a dedicated connection is allocated to an isolated session and before the isolated session user uses the connection to interact with the database.
- `preReleaseExclusiveConnection`: used to perform Oracle VPD cleanup at the time the isolated session is released and after the user is finished interacting with the database.

In the implementation of these handlers, you can obtain the required user credentials from the associated session's properties.

Using Oracle Proxy Authentication

JPA and EclipseLink are typically used in a middle tier/server environment with a shared connection pool. A connection pool allows database connections to be shared to avoid the cost of reconnecting to the database. Typically, the user logs into the application but does not have their own database login as a shared login is used for the connection pool. The provides a mechanism to set a proxy user on an existing database connection. This allows for a shared connection pool to be used, but to also gives the database a user context.

Oracle proxy authentication is configured using the following persistence unit properties on an `EntityManager` object:

- `"eclipselink.oracle.proxy-type" : oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME, PROXYTYPE_CERTIFICATE, PROXYTYPE_DISTINGUISHED_NAME`
- `oracle.jdbc.OracleConnection.PROXY_USER_NAME : `*`user_name`*
- `oracle.jdbc.OracleConnection.PROXY_USER_PASSWORD : `*`password`*
- `oracle.jdbc.OracleConnection.PROXY_DISTINGUISHED_NAME`
- `oracle.jdbc.OracleConnection.PROXY_CERTIFICATE`
- `oracle.jdbc.OracleConnection.PROXY_ROLES`

This connection is only used for writing by default; reads still use the shared connection pool. To force reads to also use the connection, the `eclipselink.jdbc.exclusive-connection.mode` property should be set to `Always`, but this depends on if the application wishes to audit writes or reads as well. The `eclipselink.jdbc.exclusive-connection.is-lazy` property configures whether the

> connection should be connected up front, or only when first required. If only writes are audited, then lazy connections allow for the cost of creating a new database connection to be avoided unless a write occurs.

Main Tasks:

To setup proxy authentication, create an `EntityManager` object and set the persistence unit properties. Three examples are provided:

Task: Audit Only Writes

To configure proxy authentication when auditing only writes:

```
Map properties = new HashMap();
properties.put("eclipselink.oracle.proxy-type",
 oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
properties.put(oracle.jdbc.OracleConnection.PROXY_USER_NAME, user);
properties.put(oracle.jdbc.OracleConnection.PROXY_USER_PASSWORD, password);
properties.put("eclipselink.jdbc.exclusive-connection.mode", "Transactional");
properties.put("eclipselink.jdbc.exclusive-connection.is-lazy", "true");
EntityManager em = factory.createEntityManager(properties);
```

Task: Audit Reads and Writes

To configure proxy authentication when auditing reads and writes:

```
Map properties = new HashMap();
properties.put("eclipselink.oracle.proxy-type",
 oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
properties.put(oracle.jdbc.OracleConnection.PROXY_USER_NAME, user);
properties.put(oracle.jdbc.OracleConnection.PROXY_USER_PASSWORD, password);
properties.put("eclipselink.jdbc.exclusive-connection.mode", "Always");
properties.put("eclipselink.jdbc.exclusive-connection.is-lazy", "false");
EntityManager em = factory.createEntityManager(properties);
```

Task: Configure Proxy Authentication in Jakarta EE Applications

If a JEE and JTA managed entity manager is used, specifying a proxy user and password can be more difficult, as the entity manager and JDBC connection is not under the applications control. The persistence unit properties can still be specified on an `EntityManager` object as long as this is done before establishing a database connection.

If using JPA 2.`n`, the `setProperty` API can be used:

```
em.setProperty("eclipselink.oracle.proxy-type",
   oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
em.setProperty(oracle.jdbc.OracleConnection.PROXY_USER_NAME, user);
em.setProperty(oracle.jdbc.OracleConnection.PROXY_USER_PASSWORD, password);
```

```
em.setProperty("eclipselink.jdbc.exclusive-connection.mode", "Always");
em.setProperty("eclipselink.jdbc.exclusive-connection.is-lazy", "false");
```

Otherwise, the `getDelegate` API can be used:

```
Map properties = new HashMap();
properties.put("eclipselink.oracle.proxy-type",
   oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
properties.put(oracle.jdbc.OracleConnection.PROXY_USER_NAME, user);
properties.put(oracle.jdbc.OracleConnection.PROXY_USER_PASSWORD, password);
properties.put("eclipselink.jdbc.exclusive-connection.mode", "Always");
properties.put("eclipselink.jdbc.exclusive-connection.is-lazy", "false");
((org.eclipse.persistence.internal.jpa.EntityManagerImpl)em.getDelegate()).
   setProperties(properties);
```

Caching and security

By default, EclipseLink maintains a shared (L2) object cache. This is fine for auditing, but if Oracle VPD or user based security is used to prevent the reading of certain tables/classes, then the cache may need to be disabled for these secure classes. To disable the shared cache, see "Disabling Entity Caching".

If the database user is used to check security for reads, then set the `eclipselink.jdbc.exclusive-connection.mode` property to `Isolated` to only use the user connection for reads for the classes whose shared cache has been disabled (isolated).

Using Oracle Virtual Private Database for Row-Level Security

The Oracle Virtual Private Database (VPD) feature allows for row level security within the Oracle database. Typically, database security only allows access privileges to be assigned per table. Row level security allows different users to have access to different rows within each table.

The Oracle proxy authentication features in EclipseLink can be used to support Oracle VPD. The proxy user allows for the row level security to be checked. When using Oracle VPD, it is also important to disable shared caching for the secured objects as these objects should not be shared. To disable the shared cache, see "Disabling Entity Caching".

Using EclipseLink with Oracle RAC

Oracle Real Application Clusters (RAC) extends the Oracle Database so that you can store, update, and efficiently retrieve data using multiple database instances on different servers at the same time. Oracle RAC provides the software that manages multiple servers and instances as a single group. Applications use Oracle RAC features to maximize connection performance and availability and to mitigate down-time due to connection problems. Applications have different availability and performance requirements and implement Oracle RAC features accordingly. For details on Oracle RAC, see the *Oracle Real Application Clusters Administration and Deployment Guide*.

The Oracle Database and the Oracle WebLogic Server both provide connection pool

implementations that can create connections to a RAC database and take advantage of various Oracle RAC features. The features include Fast Connection Failover (FCF), Run-Time Connection Load Balancing (RCLB), and connection affinity. In WebLogic Server, applications create JDBC data sources (Multi Data Source or GridLink Data Source) to connect to a RAC-enabled database. Standalone applications use the Universal Connection Pool (UCP) JDBC connection pool API (`ucp.jar`) to create data sources. Both connection pool implementations require the Oracle Notification Service library (`ons.jar`). This library is the primary means by which the connection pools register for, and listen to, RAC events. For those new to these technologies, refer to the *Oracle Universal Connection Pool for JDBC Developer's Guide* and the Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server.

This sections assumes that you have an Oracle JDBC driver and Oracle RAC-enabled database. Make sure that the RAC-enabled database is operational and that you know the connection URL. In addition, download the database Oracle Client software that contains the `ons.jar` file. The `ucp.jar` file is included with the Oracle Database.

Accessing a RAC-Enabled database from Jakarta EE Applications

The tasks in this section are used to connect to a RAC-enabled database from a persistence application implemented in Oracle WebLogic Server.

Task 1: Configure a Multi Data Source or GridLink Data Source

Refer to Chapter 3, "Using EclipseLink with WebLogic Server," and *Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server* for details about configuring a data source in WebLogic Server for Oracle RAC.

Task 2: Configure the Persistence Unit

Edit the `persistence.xml` file and include the name of the data source within a persistence unit configuration. For example:

```
<persistence-unit name="OrderManagement">
   <jta-data-source>jdbc/MyOrderDB</jta-data-source>
   ...
</persistence-unit>
```

Task 3: Include the Required JARs

Ensure that the `ons.jar` is in the WebLogic Server classpath.

Accessing a RAC-Enabled Database from Standalone Applications

The tasks in this section are used to connect to a RAC database from a standalone persistence application. The tasks demonstrate how to use UCP data sources which are required for advanced RAC features.

Task 1: Create a UCP Data Source

A UCP data source is used to connect to a RAC database. The data source can specify advanced RAC configuration. For details on using advanced RAC features with UCP, see *Oracle Universal Connection Pool for JDBC Developer's Guide*. The following example creates a data source and enables FCF and configures ONS.

```
PoolDataSource datasource = PoolDataSourceFactory.getPoolDataSource();
datasource.setONSConfiguration("nodes=host1:4200,host2:4200");
datasource.setFastConnectionFailoverEnabled(true);
datasource.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
datasource.setURL("jdbc:oracle:thin:@DESCRIPTION=
    (LOAD_BALANCE=on)
    (ADDRESS=(PROTOCOL=TCP)(HOST=host1)(PORT=1521))
    (ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))
    (ADDRESS=(PROTOCOL=TCP)(HOST=host3)(PORT=1521))
    (ADDRESS=(PROTOCOL=TCP)(HOST=host4)(PORT=1521))
    (CONNECT_DATA=(SERVICE_NAME=service_name)))");
```

Applications that do not require the advanced features provided by RAC and UCP can connect to a RAC-enabled database using the native connection pool in EclipseLink. In this case, edit the `persistence.xml` file for you applications and add the RAC URL connection string for a persistence unit. For example:

```
<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="jakarta.persistence.jdbc.driver"
            value="oracle.jdbc.OracleDriver"/>
         <property name="jakarta.persistence.jdbc.url"
            value="jdbc:oracle:thin@(DESCRIPTION= "+ "(LOAD_BALANCE=on)"+
           "(ADDRESS=(PROTOCOL=TCP)(HOST=rac_node) (PORT=1521))"+
           "(ADDRESS=(PROTOCOL=TCP)(HOST=racnode2) (PORT=1521))"+
           "(CONNECT_DATA=(SERVICE_NAME=service_name))")"/>
         <property name="jakarta.persistence.jdbc.user" value="user_name"/>
         <property name="jakarta.persistence.jdbc.password" value="password"/>
      </properties>
   </persistence-unit>
</persistence>
```

To use the persistence unit, instantiate an `EntityManagerFactory` as follows:

```
Persistence.createEntityManagerFactory("my-app");
```

Task 2: Use the UCP Data Source

To use the UCP data source, instantiate an `EntityManagerFactory` an pass in the data source as follows:

```
Map properties = new HashMap();
       properties.add("jakarta.persistence.nonJtaDataSource", datasource);
       Persistence.createEntityManagerFactory(properties);
```

Task 3: Include the Required JARs

Ensure that both `ucp.jar` and `ons.jar` are in the application classpath.

Using Oracle Spatial and Graph

EclipseLink provides added support for querying Oracle Spatial and Graph data in the Oracle Database. Oracle Spacial and Graph is used to location-enable applications. It provides advanced features for spatial data and analysis and for physical, logical, network, and social and semantic graph applications. The spatial features provide a schema and functions that facilitate the storage, retrieval, update, and query of collections of spatial features in an Oracle database. For details about developing Oracle Spacial and Graph applications, see *Oracle Spatial and Graph Developer's Guide*. To use Oracle Spatial and Graph within WebLogic Server, see Chapter 3, "Task 7: Extend the Domain to Use Advanced Oracle Database Features,"

EclipseLink applications can construct expressions that use Oracle Spacial and Graph operators. See the `org.eclipse.persistence.expressions.spatial` API for details. For Example:

```
ExpressionBuilder builder = new ExpressionBuilder();
Expression withinDistance = SpatialExpressions.withinDistance(myJGeometry1,
   myJGeometry2, "DISTANCE=10");
session.readAllObjects(GeometryHolder.class, withinDistance);
```

The above expression requires a `oracle.spatial.geometry.JGeometry` object. Use the EclipseLink `org.eclipse.persistence.platform.database.oracle.converters.JGeometryConverter` converter to convert the `JGeometry` object as it is read and written from the Oracle database. The `JGeometryConverter` object must be added to the Oracle Database platform either with the `addStructConverter(StructConverter)` method or specified in the `sessions.xml` file. The `JGeometry` type must also be available on the classpath.

The following example demonstrates how to use the `FUNCTION` JPA extension to perform Oracle Spatial queries. For details on the `FUNCTION` extension, see *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*:

```
SELECT a FROM Asset a, Geography geo WHERE geo.id = :id AND a.id IN :id_list AND
```

```
   FUNCTION('ST_INTERSECTS', a.geometry, geo.geometry) = 'TRUE'

SELECT s FROM SimpleSpatial s WHERE FUNCTION('MDSYS.SDO_RELATE', s.jGeometry,
   :otherGeometry, :params) = 'TRUE' ORDER BY s.id ASC
```

22.3. Additional Resources

See the following links for additional resources about the solutions discussed in this chapter.

- *Java API Reference for EclipseLink*
- *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*
- *Oracle Database JDBC Java API Reference*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Database Security Guide*
- *Oracle Label Security Administrator's Guide*
- *Oracle Fusion Middleware Configuring and Managing JDBC Data Sources for Oracle WebLogic Server*
- *Oracle Real Application Clusters Administration and Deployment Guide*
- *Oracle Universal Connection Pool for JDBC Developer's Guide*
- *Oracle Spatial and Graph Developer's Guide*
