Table of Contents

Understanding EclipseLink

PDF | ePub

EclipseLink

Understanding EclipseLink 4.0.5

December 2022

Understanding EclipseLink (Concepts Guide)

Print date: December, 2022

Preface

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

Audience

This document is intended for application developers and administrators who want to know more about the concepts behind EclipseLink and its features.

Related Documents

For more information, see the following documents in the EclipseLink documentation set:

- *Solutions Guide for EclipseLink*
- *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*
- *Developing JAXB Applications EclipseLink MOXy*
- *Developing Persistence Architectures Using EclipseLink Database Web Services Developer's Guide*
- Java API Reference for EclipseLink
- Java Persistence specification for complete information about JPA

  `http://jcp.org/en/jsr/detail?id=338`

Conventions

The following text conventions are used in this document:

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

List of Examples

Chapter 1. Overview of EclipseLink

This chapter describes EclipseLink and its key features: the components that are included with EclipseLink, metadata, application architectures, mappings, and the API.

This chapter includes the following sections:

1.1. Understanding EclipseLink

EclipseLink is an open-source mapping and persistence framework for use in a Java environment, including Java Platform, Standard Edition (Java SE) and Java Platform, Enterprise Edition (Jakarta EE). The EclipseLink project is under the stewardship of the Eclipse Foundation.

EclipseLink completely implements the following specifications, plus extensions to those specifications:

- **Jakarta Persistence API (JPA)**

  JPA is the Java API for object/relational mapping (ORM), where Java objects are mapped to database artifacts, for the purpose of managing relational data in Java applications. JPA includes Java Persistence Query Language (JPQL), the Java Persistence Criteria API, and the Java API and XML schema for defining object/relational mapping metadata.

  The latest version of the specification is *JSR 338: Java Persistence 2.2*. See `http://jcp.org/en/jsr/detail?id=338`.

  Some of the EclipseLink extensions to standard JPA are: **Support for mapping to nonrelational (NoSQL) databases.** Features useful in Software as a Service (SaaS) environments, including tenant isolation, extensible entities, external metadata sources. **Java API for RESTful Web Services (JAX-RS, defined in JSR 311).** Many other additional annotations, annotation extensions, Java Persistence Query Language (JPQL) extensions, JPA query customizations extensions, and persistence property extensions.

- **Java Architecture for XML Binding (JAXB)**

  JAXB is the Java API for object/XML mapping (OXM), where an XML document is bound to Java objects, based on the XML document's XSD schema. JAXB provides methods for unmarshalling (reading) XML instance documents into Java content trees, and then marshalling (writing) Java content trees back into XML instance documents. JAXB also provides a way to generate XML schema from Java objects.

  The latest version of the specification is *JSR 222: Java Architecture for XML Binding (JAXB) 2.0*.

See http://jcp.org/en/jsr/detail?id=222.

The EclipseLink JAXB implementation is part of the **EclipseLink MOXy** component, which extends EclipseLink JAXB to support JavaScript Object Notation (JSON) documents. EclipseLink supports all object/XML options when reading and writing JSON. MOXy also includes support for the older native EclipseLink object/XML API).

In addition to the implementations of the standard specifications described above, EclipseLink includes the following:

- **EclipseLink Database Web Services (DBWS)**

  DBWS is a development tool and a runtime for providing Jakarta EE-compliant, client-neutral access to relational database artifacts via a Web service. The development tool, DBWS Builder, is a command-line utility that generates the necessary deployment artifacts. (DBWS Builder is integrated into the Eclipse Dali Java Persistence toolset and into Oracle JDeveloper.) The runtime provider takes a service descriptor (along with related deployment artifacts) and realizes it as a JAX-WS 2.0 Web service. The runtime uses EclipseLink to bridge between the database and the XML SOAP Messages used by Web service clients.

- **EclipseLink Enterprise Information Services (EIS)**

  EIS is a facility for enabling the use of data stores through Java Connector Architecture (JCA) resource adapters. Using XML metadata, the interactions and their exchanged data are configured and mapped onto a domain model. The interactions data can be mapped from either the Common Client interface (CCI) or using XML schemas. This use is intended for non-relational data stores where no JDBC or SQL access is provided.

EclipseLink can be used with a wide range of Java Enterprise Edition (Jakarta EE) and Java application architectures. Use EclipseLink to design, implement, deploy, and optimize an advanced, object-persistence and object-transformation layer that supports a variety of data sources and formats, including relational databases, nonrelational (NoSQL) databases, XML, JSON, and Web Services.

EclipseLink supports Java persistence in Jakarta EE, Java SE and web containers including integration with various application servers including:

- Oracle WebLogic Server
- Oracle Glassfish Server
- JBoss Web Server
- IBM WebSphere application server
- SAP NetWeaver
- Oracle Containers for Jakarta EE (OC4J)
- Various other web containers, such as Apache Tomcat, Eclipse Gemini, IBM WebSphere CE, and SpringSource Server

EclipseLink lets you quickly capture and define object-to-data source and object-to-data representation mappings in a flexible, efficient metadata format.

The runtime lets your application exploit this mapping metadata with a simple session facade that provides in-depth support for data access, queries, transactions (both with and without an external transaction controller), and caching.

For more information about EclipseLink, see "Key Features".

What Is the Object-Persistence Impedance Mismatch?

Java-to-data source integration is a widely underestimated problem when creating enterprise Java applications. This complex problem involves more than simply reading from and writing to a data source. The data source elements include tables, rows, columns, and primary and foreign keys. The Java and Jakarta EE programming languages include entity classes (regular Java classes), business rules, complex relationships, and inheritance. In a nonrelational data source, you must match your Java entities with XML elements and schemas.

A successful solution requires bridging these different technologies and solving the object-persistence impedance mismatch—a challenging and resource-intensive problem. To solve this problem, you must resolve the following issues between Jakarta EE and the data source elements:

- Fundamentally different technologies
- Different skill sets
- Different staff and ownership for each of the technologies
- Different modeling and design principles

As an application developer, you need a product that lets you integrate Java applications with any data source, without compromising application design or data integrity. In addition, as a Java developer, you need the ability to store (that is, persist) and retrieve business domain objects using a relational database or a nonrelational data source as a repository.

The EclipseLink Solution

EclipseLink addresses the disparity between Java objects and data sources. It contains a persistence framework that lets you build applications that combine the best aspects of object technology with a specific data source. You can do the following:

- Persist Java objects to virtually any relational database
- Perform in-memory conversions between Java objects and XML and JSON documents
- Map any object model to any relational or nonrelational schema
- Use EclipseLink successfully, even if you are unfamiliar with SQL or JDBC, because EclipseLink offers a clear, object-oriented view of data sources

1.2. Key Features

An extensive set of features are provided. You can use these features to rapidly build high-performance enterprise applications that are scalable and maintainable.

Some of the primary features are the following:

- Nonintrusive, flexible, metadata-based architecture
- Advanced mapping support and flexibility: relational, object-relational data type, and XML
- Optimized for highly scalable performance and concurrency with extensive performance tuning options
- Comprehensive object caching support including cluster integration for some application servers (such as Oracle Fusion Middleware Server)
- Extensive query capability including: Java Persistence Query Language (JPQL), native SQL, and EclipseLink Expressions framework
- Just-in-time reading
- Object-level transaction support and integration with popular application servers and databases
- Optimistic and pessimistic locking options and locking policies

For additional information and downloads, see the EclipseLink home page:

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

1.3. Key Concepts

This section provides a brief introduction to several of the key concepts described in this documentation. The key concepts highlighted in this section are as follows:

- EclipseLink Metadata
- Entities
- Descriptors
- Mappings
- Data Access
- Caching
- Queries
- Expression Framework
- NoSQL Databases
- Performance Monitoring and Profiling

EclipseLink Metadata

EclipseLink metadata is the bridge between the development of an application and its deployed runtime environment. You can capture the metadata using:

- JPA annotations in Java files and the JPA-defined properties in the `persistence.xml` and `eclipselink-orm.xml` files.

  Metadata is also captured by EclipseLink JPA annotations and property extensions in the `persistence.xml` file. The `eclipselink-orm.xml` file can also be used to specify property extensions beyond the JPA specification.

- JAXB annotations in Java files and JAXB-defined properties in the `eclipselink-oxm.xml` file.

  The `eclipselink-oxm.xml` file can be used to define property extensions beyond the JAXB specification.

- Java and the EclipseLink API.

The metadata lets you pass configuration information into the runtime environment. The runtime environment uses the information in conjunction with the persistent classes, such as Java objects, JPA entities, and the code written with the EclipseLink API, to complete the application. See "Adding Metadata Using Annotations" for more information. See also *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Mappings can be stored external to the application. This can be as simple as making the `eclipselink-orm.xml` or `eclipselink-oxm.xml` file with the additional mapping information available on a web server as a file. It can also be more complex involving a server process that stores the mapping information and allows the information to be updated dynamically. For more information, see "EclipseLink/Examples/JPA/MetadataSource" in the EclipseLink documentation.

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

Entities

An entity is a persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in the table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

See Chapter 4, "Understanding Entities."

Descriptors

Descriptors describe how a Java class relates to a data source representation. They relate object classes to the data source at the data model level. For example, persistent class attributes may map to database columns.

EclipseLink uses descriptors to store the information that describes how an instance of a particular class can be represented in a data source. Descriptors are used internally by EclipseLink, and are defined through annotations, XML, or in IDEs such as JDeveloper or Eclipse, then read at run time.

See Chapter 5, "Understanding Descriptors."

Mappings

Mappings describe how individual object attributes relate to a data source representation. Mappings can involve a complex transformation or a direct entry.

EclipseLink uses mappings to determine how to transform data between object and data source representations. Mappings are used internally by EclipseLink, and are defined through annotations, XML, or in IDEs such as Eclipse, then read from XML files at run time.

See Chapter 6, "Understanding Mappings."

Data Access

A data source platform includes options specific to a particular data source including binding, use of native SQL, use of batch writing, and sequencing.

See Chapter 7, "Understanding Data Access."

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 is used to improve performance by avoiding unnecessary trips to the database, manage locking and cache isolation level, and manage object identity.

See Chapter 8, "Understanding Caching."

Queries

You can to create, read, update, and delete persistent objects or data using queries in both Jakarta EE and non-Jakarta EE applications for both relational and nonrelational data sources. Queries can

be made at the object level or data level.

A number of query languages are supported, such as Java Persistence Query Language (JPQL), SQL, and the Expression Framework. The Java Persistence Criteria API can also be used to define dynamic queries through the construction of object-based query definition objects, rather than use of the string-based approach of JPQL.

See Chapter 9, "Understanding Queries."

Expression Framework

By using the EclipseLink Expressions framework, you can specify query search criteria based on your domain object model. Expressions offer a number of advantages over SQL. For example, expressions are easier to maintain, changes to descriptors or database tables do not affect the querying structures in the application, they enhance readability by standardizing the `Query` interface, and they simplify complex operations.

See Chapter 10, "Understanding EclipseLink Expressions."

NoSQL Databases

NoSQL is a classification of database systems that do not support the SQL standard. These include document databases, key-value stores, and various other non-standard databases. Persistence of Java objects to NoSQL databases is supported through the Jakarta Persistence API (JPA). EclipseLink's native API is also supported with NoSQL databases.

See Chapter 11, "Understanding Non-relational Data Sources."

Performance Monitoring and Profiling

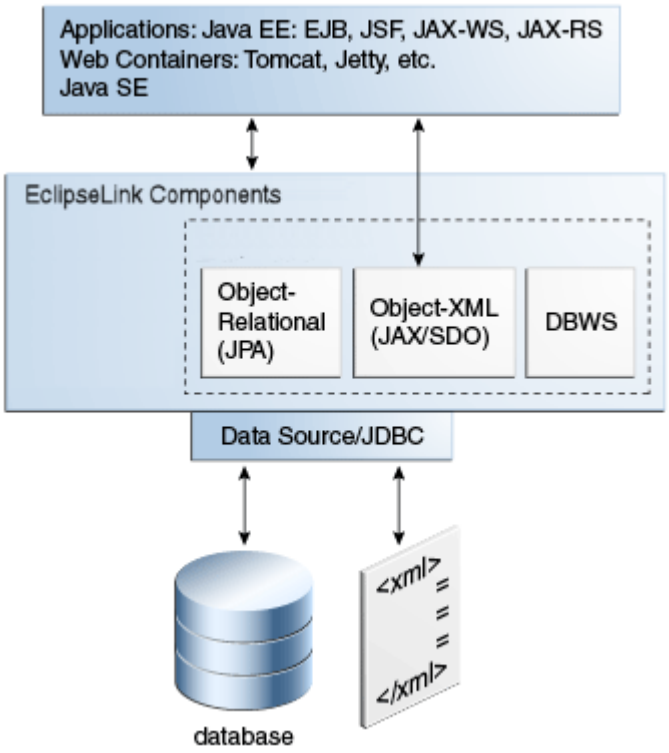
A diverse set of features is provided to measure and optimize application performance. You can enable or disable most features in the descriptors or session, making any resulting performance gains global. Tools are provided for performance profiling and performance, fetch group, and query monitoring.

See "Enhancing Performance" in *Solutions Guide for EclipseLink*.

1.4. Key Components

Figure 1-1 illustrates the components contained by EclipseLink. The following sections describe the components.

Figure 1-1 EclipseLink Components



Description of "Figure 1-1 EclipseLink Components"

EclipseLink Core and API

The EclipseLink Core provides the runtime component. Access to the runtime can be obtained directly through the EclipseLink API. The runtime environment is not a separate or external process—it is embedded within the application. Application calls invoke EclipseLink to provide persistence behavior. This function enables transactional and thread-safe access to shared database connections and cached objects.

The EclipseLink API provides the reference implementation for JPA 2.0 (JSR-338). The `org.eclipse.persistence.*` classes encapsulate the EclipseLink API and provide extensions beyond the specification. These extensions include EclipseLink-specific properties and annotations. For more information on the API, properties and extensions, see *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

The JAXB APIs are included in Java SE 6. In the `eclipselink.jar` file, the `org.eclipse.persistence.jaxb.*` classes encapsulate the EclipseLink support for JAXB.

Object-Relational (JPA 2.2) Component

JPA simplifies Java persistence. It provides an object-relational mapping approach that lets you declaratively define how to map Java objects to relational database tables in a standard, portable way. JPA works both inside a Jakarta EE application server and outside an EJB container in a Java Standard Edition (Java SE) application. The main features included in the 2.2 JPA update are:

- Expanded object/relational mapping functionality
  - Support for collections of embedded objects

    - Multiple levels of embedded objects
    - Ordered lists
    - Combinations of access types
- A criteria query API
- Standardization of query "hints"
- Standardization of additional metadata to support DDL generation
- Support for validation

JAXB Component

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`

MOXy Component

MOXy (also known as Object-XML) is the EclipseLink JAXB implementation. This component enables you to bind Java classes to XML schemas. MOXy implements JAXB which lets you provide mapping information through annotations. Support for storing the mappings in XML format is provided by MOXy. The many advanced mappings that are available enable you to handle complex XML structures without having to mirror the schema in your Java class model.

The objects produced by the EclipseLink JAXB compiler are Java POJO models. They are generated with the necessary annotations required by the JAXB specification. The JAXB runtime API can be used to marshal and unmarshal objects.

When using MOXy as the JAXB provider, no metadata is required to convert your existing object model to XML. You can supply metadata (using annotations or XML) only when you must fine-tune the XML representation of the model.

Using EclipseLink MOXy, you can manipulate XML in the following ways:

- Generate a Java Model from an XML schema
- Specify the EclipseLink MOXy JAXB runtime
- Use JAXB to manipulate XML
- Generate an XML schema from a Java model

For more information on MOXy and these use cases, see *Developing JAXB Applications EclipseLink MOXy*.

EclipseLink provides maximum flexibility with the ability to control how your object model is mapped to an XML schema. There are many advantages to having control over your own object model:

- You can design the domain classes specifically for your application using the appropriate patterns and practices.
- You can use XPath-based mapping. This prevents the need for having a 1-to-1 relationship between classes and XML schema types. For more information, see *Developing JAXB Applications EclipseLink MOXy*.
- You can instantiate objects in a way that is appropriate to your application.
- You can control your own class path dependencies. Most JAXB implementations put vendor specific code in the generated classes that add class path dependencies to your application.

One of the key advantages of EclipseLink is that the mapping information can be stored externally and does not require any changes to the Java classes or XML schema. This means that you can map your domain objects to more than one schema, or if your schema changes, you can update the mapping metadata instead of modifying your domain classes. This is also useful when mapping third-party classes, because you might not have access to the source to add annotations.

SDO Component

The Service Data Objects (SDO) component provides the reference implementation of Service Data Objects version 2.1.1. The reference implementation is described in JSR-235. The SDO implementation incorporates the reference implementation and provides additional features primarily used for converting Java objects to XML, and for building and using data object models that can be incorporated into service architectures.

SDO provides you with the following capabilities:

- Use of the SDO APIs
- Conversion an XML Schema
- Customization of your XSD for SDO usage
- Use of dynamic data objects to manipulate XML
- Use of static data objects
  - Run the SDO compiler—generate type safe data objects
  - Use type safe data objects to manipulate XML

For more information, see "Getting Started with EclipseLink SDO" in the EclipseLink documentation:

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

Database Web Services Component

Database Web Services (DBWS) enables simple and efficient access to relational database artifacts by using a web service. It provides Jakarta EE-compliant client-neutral access to the database without having to write Java code. DBWS extends EclipseLink's core capabilities while using existing ORM and OXM components.

DBWS has a runtime provider component that takes a service descriptor (along with related deployment artifacts) and realizes it as a JAX-WS 2.0 Web service. The runtime provider uses EclipseLink to bridge between the database and the XML SOAP Messages used by web service clients. For information on DBWS architecture, see *Developing Persistence Architectures Using EclipseLink Database Web Services Developer's Guide*.

1.5. Key Tools

This section describes the support for EclipseLink provided by Oracle JDeveloper, Eclipse, and NetBeans development environments. Any tooling is compatible with EclipseLink, but these offer specific integration.

This section contains the following subsections:

- Oracle JDeveloper
- Eclipse
- NetBeans

Oracle JDeveloper

Oracle JDeveloper is a Jakarta EE development environment with end-to-end support to develop, debug, and deploy e-business applications and web services.

For JDeveloper information and downloads, see:

`http://www.oracle.com/us/products/tools/019657.htm`

JDeveloper includes a number of features to aid in the development of applications that use EclipseLink. These features include wizards to reverse engineer JPA entities from database tables and to generate EJB 3.0 Session Beans with `EntityManager` injection. It also includes methods for querying JPA entities and to test client generation.

JDeveloper tools enable you to quickly and easily configure and map your Java classes and JPA entities to different data sources, including relational databases and XML schemas without using

code.

DBWSBuilder script (included in EclipseLink install) can be used to run the DBWSBuilder utility to generate the Web service. JDeveloper uses the API provided, but does not use the DBWSBuilder script directly.

For more information on Oracle JDeveloper's EclipseLink support, see JDeveloper online help.

Eclipse

The Eclipse IDE provides a number of features and utilities to help you create, run, and maintain applications that use JPA. These capabilities are extended if you install OEPE.

For Eclipse IDE information and downloads, see:

`http://www.oracle.com/technetwork/developer-tools/eclipse/overview/index.html`

The Dali Java Persistence Tools Project provides extensible frameworks and tools for defining and editing object-relational mappings for JPA entities. JPA mapping support focuses on minimizing the complexity of mapping by providing entity generation wizards, design-time validation, and a robust UI for entity and persistence unit configuration.

For Dali information and downloads, see:

`http://www.eclipse.org/webtools/dali`

Other tools and utilities from the Oracle, open source, and third party vendor communities are available from the Eclipse Marketplace.

`http://marketplace.eclipse.org/`

NetBeans

NetBeans IDE bundles Oracle GlassFish Server, which includes EclipseLink. The IDE provides full support for JPA-based code development. This support includes entity class wizards for constructing entities and editor hints to ensure that entities conform to the JPA specification. NetBeans also provides a persistence unit editor for constructing a `persistence.xml` file.

For NetBeans information and downloads, see:

`http://netbeans.org/index.html`

Chapter 2. Understanding Object-Relational and MOXy Internals

This chapter describes the features of the Object-Relational and MOXy architecture.

This chapter includes the following sections:

- About Metadata
- About the Object-Relational Solution
- About the MOXy Solution

2.1. About Metadata

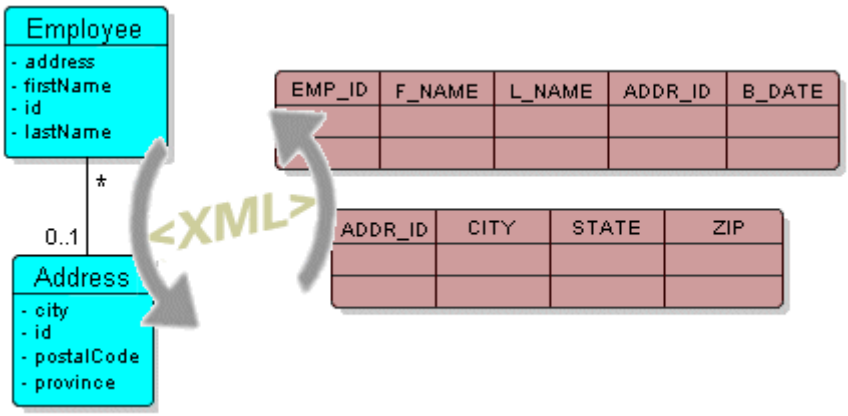
The EclipseLink metadata is the bridge between the development of an application and its deployed run-time environment. Capture the metadata using the following:

- Annotations, and elements property extensions in the `persistence.xml` file. The persistence provider interprets all these sources of metadata to create an in-memory session and project at run time.
- Java and the EclipseLink API (this approach is the most labor-intensive).

The metadata lets you pass configuration information into the run-time environment. The run-time environment uses the information in conjunction with the persistent classes (Java objects or entities) and the code written with the EclipseLink API, to complete the application.

For more information, see "Overriding and Merging" in Jakarta Persistence API (JPA) Extensions Reference for EclipseLink.

Figure 2-1 EclipseLink Metadata



Description of "Figure 2-1 EclipseLink Metadata"

This section describes the following:

- Advantages of the Metadata Architecture

- Creating Project Metadata
- Creating Session Metadata

Advantages of the Metadata Architecture

The EclipseLink metadata architecture provides many important benefits, including the following:

- Stores mapping information in XML—not in the domain model objects
- By using the metadata, EclipseLink does not intrude in the object model or the database schema
- Allows you to design the object model as needed, without forcing any specific design
- Allows DBAs to design the database as needed, without forcing any specific design
- Does not rely on code-generation (which can cause serious design, implementation, and maintenance issues)
- Is unobtrusive: adapts to the object model and database schema, rather than requiring you to design their object model or database schema to suit EclipseLink

Using EclipseLink JPA, you have the flexibility of expressing persistence metadata using standard JPA annotations, deployment XML, or both and you can optionally take advantage of EclipseLink JPA annotation and `persistence.xml` property extensions.

Creating Project Metadata

A project contains the mapping metadata that the EclipseLink runtime uses to map objects to a data source. The project is the primary object used by the EclipseLink runtime.

This section describes the principal contents of project metadata, including the following:

- Entity Mappings
- Data Source Login Information

For Object-relational mapping, the EclipseLink runtime constructs an in-memory project based on any combination of JPA annotations, `persistence.xml`, and EclipseLink JPA annotations and `persistence.xml` property extensions.

For MOXy mapping, the EclipseLink runtime uses a combination of JAXB annotations and `eclipselink-oxm` bindings. See "Overriding and Merging" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Entity Mappings

EclipseLink maps persistent entities to the database in the application. There are several approaches to project development, including the following:

- Importing classes and tables for mapping
- Importing classes and generating tables and mappings
- Importing tables and generating classes and mappings
- Creating both class and table definitions

The most common solution is to develop the persistent entities using a development tool, such as a modeling tool or an integrated development environment (IDE) like JDeveloper, and to develop the relational model through appropriate relational design tools. You then use JDeveloper to construct mappings that relate these two models.

Although JDeveloper offers the ability to generate persistent entities or the relational model components for an application, these utilities are intended only to assist in rapid initial development strategies—not complete round-trip application development.

For more information, see Chapter 5, "Understanding Descriptors" and Chapter 6, "Understanding Mappings".

Data Source Login Information

For POJO projects, you configure a session login in the session metadata that specifies the information required to access the data source.

For more information, see Creating Session Metadata.

Creating Session Metadata

A EclipseLink session contains the information required to access the data source. The session is the primary object used by your application to access the features of the EclipseLink runtime.

Using EclipseLink JPA, the EclipseLink runtime constructs an in-memory session based on any combination of JPA and EclipseLink annotations, JPA persistence properties (in the `persistence.xml` file), and EclipseLink property extensions. See "Overriding and Merging" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

2.2. About the Object-Relational Solution

EclipseLink provides a complete, JPA-compliant JPA implementation. It provides complete compliance for all of the mandatory features and many of the optional features. It also supports EclipseLink features not described in the JPA specification, such as object-level cache, distributed cache coordination, extensive performance tuning options, enhanced Oracle Database support, advanced mappings, optimistic and pessimistic locking options, extended annotations, and query hints.

For more information, see *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

The following sections describe many of these features.

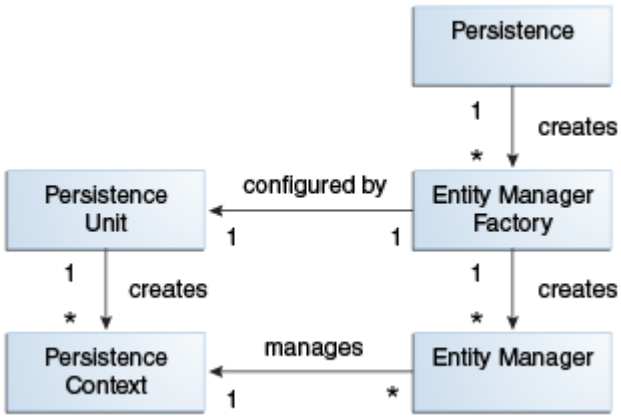
- Understanding Object-Relational Entity Architecture
- Adding Metadata Using Annotations
- About Configuration Basics
- About Data Sources
- About EclipseLink Caches
- About Database Queries

Understanding Object-Relational Entity Architecture

The entity architecture is composed of entities, persistence units, persistence contexts, entity manager factories, and entity managers. Figure 2-2 illustrates the relationships between these elements:

- Persistence creates one or more `EntityManagerFactory` objects.
- Each `EntityManagerFactory` is configured by one persistence unit.
- `EntityManagerFactory` creates one or more `EntityManager` objects.
- One or more `EntityManagers` manage one `PersistenceContext`.

Figure 2-2 Relationships Between Entity Architecture Elements



Description of "Figure 2-2 Relationships Between Entity Architecture Elements"

Entities

An entity is any application-defined object with the following characteristics:

- It can be made persistent.
- It has a persistent identity (a key that uniquely identifies an entity instance and distinguishes it from other instances of the same entity type. An entity has a persistent identity when there is a representation of it in a data store).

- It is transactional in a sense that a *persistence view* of an entity is transactional (an entity is created, updated, and deleted within a transaction, and a transaction is required for the changes to be committed in the database). However, in-memory entities can be changed without the changes being persisted.
- It is *not* a primitive, a primitive wrapper, or built-in object. An entity is a fine-grained object that has a set of aggregated states that is typically stored in a single place (such as a row in a table) and have relationships to other entities.

The entity also contains entity metadata that describes the entity. Entity metadata is not persisted to the database. It is used by the persistence layer to manage the entity from when it is loaded until it is invoked at runtime. Metadata can be expressed as annotations on the Java programming elements or in XML files (descriptors). For more information, see Chapter 4, "Understanding Entities."

Beginning with the current release, you can define and use extensible entities where mappings can be added spontaneously. In this case, the entity stores extended attributes within a map instead of static attributes. The entity then defines how values from this map are mapped to the database using an `eclipselink-orm.xml` mapping file. In addition to being able to dynamically define mappings, EclipseLink also enables these extended mappings to be stored and managed externally. This external storage enables your extended mappings to be defined while the application is running. For more information on making entities extensible, see "Providing Software as a Service" in *Solutions Guide for EclipseLink*.

Persistence and Persistence Units

Persistence is a characteristic of an entity. This means that the entity can be represented in a data store, and it can be accessed at a later time.

A persistence unit identifies a persistable unit and defines the properties associated with it. It also defines the objects that must be persisted. The objects can be entity classes, embeddable classes, or mapped superclasses. The persistence unit provides the configuration for the entity manager factory. Entity managers created by the entity manager factory inherit the properties defined in the persistence unit.

Entity Managers

An entity manager enables API calls to perform operations on an entity. Until an entity manager is used to create, read, or write an entity, the entity is a nonpersistent Java object. When an entity manager obtains a reference to an entity, that entity becomes managed by the entity manager. The set of managed entity instances within an entity manager at any given time is called its persistence context; only one Java instance with the same persistent identity can exist in a persistence context at any time.

You can configure an entity manager to read or write to a particular database, to persist or manage certain types of objects, and to be implemented by a specific persistence provider. The persistence provider supplies the implementation for JPA, including the `EntityManager` interface implementation, the Query implementation, and the SQL generation.Entity managers are provided by an `EntityManagerFactory`. The configuration for an entity manager is bound to the `EntityManagerFactory`, but it is defined separately as a persistence unit. You name persistence units

to enable differentiation between `EntityManagerFactory` objects. This way, your application obtains control over which configuration to use for operations on a specific entity. The configuration that describes the persistence unit is defined in a `persistence.xml` file. You name persistence units to be able to request a specific configuration to be bound to an `EntityManagerFactory`.

Adding Metadata Using Annotations

An annotation is a simple, expressive means of decorating Java source code with metadata that is compiled into the corresponding Java class files for interpretation at run time by a JPA persistence provider to manage persistent behavior.

A metadata annotation represents a Java language feature that lets you attach structured and typed metadata to the source code. Annotations alone are sufficient for the metadata specification—you do not need to use XML. Standard JPA annotations are in the `jakarta.persistence` package.

For more information, see Chapter 10 "Metadata Annotations" in the JPA Specification `http://jcp.org/en/jsr/detail?id=338`

EclipseLink provides a set of proprietary annotations as an easy way to add metadata to the Java source code. The metadata is compiled into the corresponding Java class files for interpretation at run time by a JPA persistence provider to manage persistent behavior. You can apply annotations at the class, method, and field levels.

EclipseLink annotations expose some features that are currently not available through the use of JPA metadata:

- Basic properties—By default, the EclipseLink persistence provider automatically configures a basic mapping for simple types. Use these annotations to fine-tune the immediate state of an entity in its fields or properties.
- Relationships—EclipseLink has defaults for some relationships, such as One-To-One and One-To-Many. Other relationships must be mapped explicitly. Use the annotations to specify the type and characteristics of entity relationships and to fine-tune how your database implements these relationships.
- Embedded objects—An embedded object does not have its own persistent identity; it is dependent upon an entity for its identity. By default, the persistence provider assumes that every entity is mapped to its own table. Use annotations to override this behavior for entities that are owned by other entities.

Advantages and Disadvantages of Using Annotations

Using annotations provides several advantages:

- They are relatively simple to use and understand.
- They provide in-line metadata within with the code that it describes; you do not need to replicate the source code context of where the metadata applies.

The primary disadvantage of annotations is that the metatdata becomes unnecessarily coupled to

the code; changes to metadata require changing and recompiling the source code.

About Configuration Basics

The following sections describe some of the key configuration files in an Object Relational Mapping project.

Default Annotation Values

Each annotation has a default value (consult the JPA specification for defaults). A persistence engine defines defaults that apply to the majority of applications. You only need to supply values when you want to override the default value. Therefore, having to supply a configuration value is not a requirement, but the exception to the rule. This is known as configuration by exception.

You should be familiar with the defaults to be able to change the behavior when necessary.

The default values are described in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*. See also Chapter 10, "Metadata Annotations" in the JPA specification.

`http://jcp.org/en/jsr/detail?id=338`

The configuration is done by exception: if a value is *not* specified in one of the configuration files, then a default value is used.

Configuring Persistence Units Using persistence.xml

A persistence unit defines the details that are required when you acquire an entity manager. You specify a persistence unit by name when you acquire an entity manager factory. Use the JPA persistence file, `persistence.xml`, to configure a persistence unit. You can specify any vendor-specific extensions in the file by using a `<properties>` element.

This file appears in the `META-INF/` directory of your persistence unit JAR file or in the classpath.

For more information, see About the Persistence Unit. See also "Persistence Property Extensions Reference" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Object-Relational Data Type Mappings

Object-relational data type mappings transform certain object data member types to structured data source representations optimized for storage in specialized object-relational databases such as Oracle Database. Object-relational data type mappings let you map an object model into an object-relational model. You can use only object-relational data type mappings with specialized object-relational databases optimized to support object-relational data type data source representations.

For more information, see *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

About Data Sources

An important part of the definition of the persistence unit is the location where the provider can find data to read and write. This is called the **data source**. The data source is typically a database. The database location is specified in the form of a JDBC data source in the JNDI namespace of the server.

Typically, applications that use EclipseLink are run in the context of a JTA transaction. Specify the name of the data source in the `jta-data-source` element in the `persistence.xml` file. If the application is not run in the context of a transaction, then it is considered to be *resource-local*. In this case, specify the name of the data source in the `non-jta-data-source` element.

You can also specify a non-relational database data source, such as an XML schema.

For more information, see Chapter 7, "Understanding Data Access."

Applications can be run in standalone, or *Java SE*, mode. In this mode, the application runs outside the server, with a non-JTA compliant data source, and in a non-Oracle stack. In this case, you must provide driver-specific information, such as the JDBC driver class, the URL that the client uses to connect to the database, and the user name and password to access the database. For more information and an example of running an application in standalone mode, see "Testing EclipseLink JPA Outside a Container" in *Solutions Guide for EclipseLink*.

About EclipseLink Caches

By default, EclipseLink uses a shared object cache that caches a subset of all objects read and persisted for the persistence unit. The shared cache differs from the local `EntityManager` cache. The shared cache exists for the duration of the persistence unit (`EntityManagerFactory` or server) and is shared by all `EntityManagers` and users of the persistence unit. The local `EntityManager` cache is not shared and only exists for the duration of the `EntityManager` or transaction.

The benefit of the shared cache is that after an object is read, the database does not need to be accessed if the object is read again. Also, if the object is read by using a query, it does not need to be rebuilt, and its relationships do not need to be fetched again.

The limitation of the shared cache is that if the database is changed directly through JDBC, or by another application or server, the objects in the shared cache will be stale.

EclipseLink offers several mechanism to deal with stale data including:

- Refreshing
- Invalidation
- Optimistic locking
- Cache coordination

- Database Change Notification (DCN)

The shared cache can also be disabled, or it can be selectively enabled and disabled by using the `@Cache` or `@Cacheable` annotations.EclipseLink also offers several different caching strategies, to configure how many objects are cached and how much memory is used.

If the application detects that the cache is out of date, it can clear, refresh, or invalidate it programmatically. Clearing the cache can cause object identity issues if any of the cached objects are in use, so invalidating is safer. If you know that none of the cached objects are in use, then you can clear the cache.

For more information, see Chapter 8, "Understanding Caching."

Defining Cache Behavior

EclipseLink provides an `@Cache` annotation which lets you define cache properties. The properties include cache type, size, and refresh rules, among others. See *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Caching in Clustered Environments

Caching in a clustered environment can have problems because changes made on one server are not reflected on objects cached in other servers. This is not a problem for read-only objects, but it is for objects that are frequently updated.

EclipseLink offers several solutions to this problem.

- The cache can be disabled for the classes that frequently change.
- Cache coordination can be used to broadcast changes between the servers in the cluster to update or invalidate changed objects.
- Cache invalidation based on time-to-live or time-of-day.
- Optimistic locking prevents updates to stale objects and triggers the objects to be invalidated in the cache.

For more information, see Clustering and Cache Coordination.

About Database Queries

The object-relational component of EclipseLink supports a variety of queries.

- JPQL queries
- SQL queries
- Criteria API queries
- Native SQL queries
- EclipseLink JPA query hints

- Query casting
- Oracle Extensions for queries
- Advanced EclipseLink native queries

For information on these queries, see Chapter 9, "Understanding Queries."

2.3. About the MOXy Solution

The MOXy component, supplied by EclipseLink, enables you to efficiently bind Java classes to XML schemas. MOXy implements JAXB, enabling you to provide your mapping information through annotations and providing support for storing the mappings in XML format.

JAXB (Java Architecture for XML Binding—JSR 222) is the standard for XML Binding in Java. JAXB covers 100 percent of XML Schema concepts. EclipseLink provides a JAXB implementation with many extensions.

When using EclipseLink MOXy as the JAXB provider, no metadata is required to convert your existing object model to XML. You can supply metadata (using annotations or XML) if you want to fine-tune the XML representation.

MOXy includes many advanced mappings that let you handle complex XML structures without having to mirror the schema in your Java class model.

For more information, see *Developing JAXB Applications EclipseLink MOXy*.

The following sections describe many of these features.

- Using EclipseLink MOXy as the JAXB Provider
- Understanding MOXy Architecture
- Serving Metadata for MOXy
- About XML Bindings
- Querying Objects by XPath

Using EclipseLink MOXy as the JAXB Provider

To use MOXy as your JAXB provider, you must identify the entry point to the JAXB runtime. This entry point is the EclipseLink `JAXBContextFactory` class.

To identify the entry point, create a text file called `jaxb.properties` and enter the path to the `JAXBContextFactory` class as the value of the `jakarta.xml.bind.context.factory` context parameter, for example:

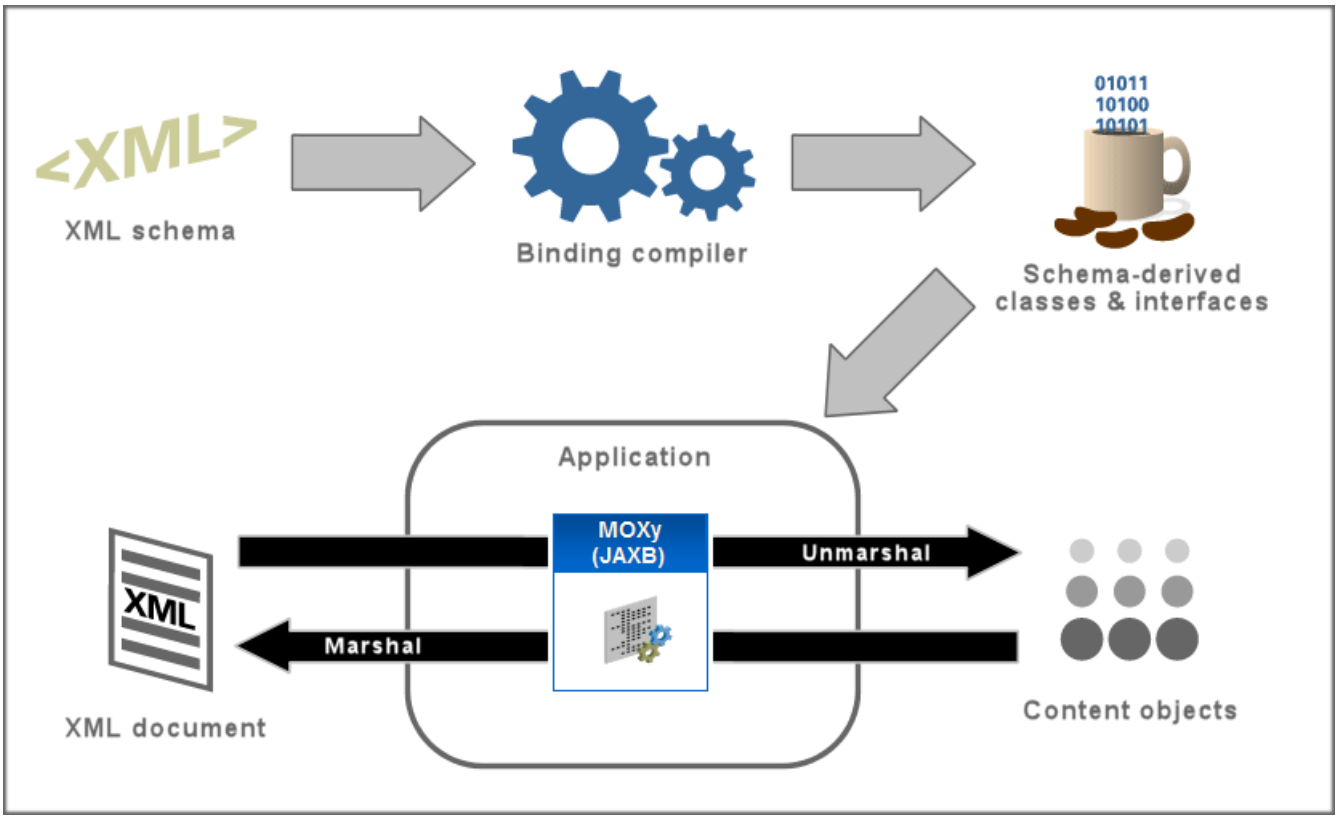
```
jakarta.xml.bind.context.factory=org.eclipse.persistence.jaxb.JAXBContextFactory
```

The `jaxb.properties` file must appear in the same package as the domain classes.

Understanding MOXy Architecture

In the sample MOXy architecture illustrated in Figure 2-3, the starting point is an XML schema. A binding compiler binds the source schema to a set of schema-derived program classes and interfaces. JAXB-annotated classes within the application are generated either by a schema compiler or the result of a developer adding JAXB annotations to existing Java classes. The application can either marshal data to an XML document or unmarshal the data to a tree of content objects. Each content object is an instance of either a schema derived or an existing program element mapped by the schema generator and corresponds to an instance in the XML.

Figure 2-3 A Sample MOXy Architecture



Description of "Figure 2-3 A Sample MOXy Architecture"

JAXB Contexts and JAXB Context Factories

The `JAXBContextFactory` class is the entry point into the EclipseLink JAXB runtime. It provides the required factory methods and can create new instances of `JAXBContext` objects.

The `JAXBContextFactory` class has the ability to:

- Create a `JAXBContext` object from an array of classes and a properties object

- Create a `JAXBContext` object from a context path and a classloader

The `JAXBContext` class provides the client's entry point to the JAXB API. The `JAXBContext` class is responsible for interpreting the metadata, generating schema files, and for creating instances of these JAXB objects: `Marshaller`, `Unmarshaller`, `Binder`, `Introspector`, and `Validator`.

MOXy offers several options when creating the `JAXBContext` object. You have the option of booting from:

- A list of one or more JAXB-annotated classes
- A list of one or more EclipseLink XML Bindings documents defining the mappings for your Java classes
- A combination of classes and XML Bindings
- A list of context paths

Serving Metadata for MOXy

In addition to the input options described in JAXB Contexts and JAXB Context Factories, MOXy provides the concept of a `MetadataSource` object. This object lets you to store mapping information outside of your application and retrieve it when the application's `JAXBContext` object is being created or refreshed. For information on implementing `MetadataSource`, see *Developing JAXB Applications EclipseLink MOXy*.

About XML Bindings

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.

EclipseLink enables you to use all of the standard JAXB annotations. In addition to the standard annotations, EclipseLink offers another way of expressing your metadata—the EclipseLink XML Bindings document. Not only can XML Bindings separate your mapping information from your actual Java class, it can also be used for more advanced metadata tasks such as:

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

For more information, see *Developing JAXB Applications EclipseLink MOXy*.

Querying Objects by XPath

In addition to using conventional Java access methods to get and set your object's values, EclipseLink MOXy also lets you access values using an XPath statement. There are special APIs on EclipseLink's `JAXBContext` object that enable you to get and set values by XPath. For more information, see *Developing JAXB Applications EclipseLink MOXy*.

Chapter 3. Understanding the Persistence Unit

This chapter describes details about the persistence unit and the persistence layer.

This chapter includes the following sections:

- About the Persistence Unit
- Building and Using the Persistence Layer
- About Persisting Objects
- Migrating Applications to the EclipseLink Persistence Manager
- About Weaving

3.1. About the Persistence Unit

A persistence unit defines the details that are required when you acquire an entity manager. To package your EclipseLink JPA application, you must configure the persistence unit during the creation of the `persistence.xml` file. Define each persistence unit in a `persistence-unit` element in the `persistence.xml` file.

Use the `persistence.xml` file to package your entities. Once you choose a packaging strategy, place the `persistence.xml` file in the `META-INF` directory of the archive of your choice. The following sections provide more detail on how to specify the persistence unit. For more information and examples, see "persistence.xml file" in the JPA Specification. For information on EclipseLink extensions to the `persistence.xml` file, see "Persistence Property Extensions Reference" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

- About the Persistence Unit Name
- About the Transaction Type Data Source
- About Mapping Files
- About Managed Classes
- About Vendor Properties
- About the Deployment Classpath
- About Persistence Unit Packaging Options
- About the Persistence Unit Scope
- About Composite Persistence Units

About the Persistence Unit Name

If you are developing your application in a Jakarta EE environment, ensure that the persistence unit name is unique within each module. For example, you can define only one persistence unit

with the name `EmployeeService` in an `emp_ejb.jar` file.

For more information, see "name" in the JPA Specification.

About 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`.

About the Transaction Type Data Source

If you are developing your application in a Jakarta EE environment, use the default transaction type: `JTA` (for Java Transaction API) and specify the data source in a `jta-data-source` element.

If you are using a data source that does not conform to the JTA, then 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, see "transaction-type" and "provider" in the JPA Specification.

About 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"/>
```

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*.

About Vendor Properties

The last section in the `persistence.xml` file is the `properties` section. The `properties` element gives you the chance to supply EclipseLink persistence provider-specific settings for the persistence unit. See "properties" in the JPA Specification. see also "Persistence Property Extensions Reference" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

About Mapping Files

Apply the metadata to the persistence unit. This metadata is a union of all the mapping files and the annotations (if there is no `xml-mapping-metadata-complete` element). If you use one mapping `orm.xml` file for your metadata, and place this file in a `META-INF` directory on the classpath, then you do not need to explicitly list it. The EclipseLink persistence provider will automatically search for this file and use it. If you named your mapping files differently or placed them in a different location, then you must list them in the mapping-file elements in the `persistence.xml` file.

For more information, see "mapping-file, jar-file, class, exclude-unlisted-classes" in the JPA Specification.

About Managed Classes

Typically, you put all of the entities and other managed classes in a single JAR file, along with the `persistence.xml` file in the `META-INF` directory, and one or more mapping files (when you store metadata in XML).

At the time the EclipseLink persistence provider processes the persistence unit, it determines which set of entities, mapped superclasses, embedded objects, and converters each particular persistence

unit will manage.

At deployment time, the EclipseLink persistence provider may obtain managed classes from any of the following sources. A managed class will be included if it is one of the following:

- Local classes: the classes annotated with `@Entity`, `@MappedSuperclass`, `@Embeddable`, or `@Converter` in the deployment unit in which its `persistence.xml` file was packaged. For more information, see "Entity" in the JPA Specification.

If you are deploying your application in the Jakarta EE environment, the application server itself, not the EclipseLink persistence provider, will discover local classes. In the Java SE environment, you can use the `exclude-unlisted-classes` element to `false` to enable this functionality—EclipseLink persistence provider will attempt to find local classes if you set this element to false. See "mapping-file, jar-file, class, exclude-unlisted-classes" in the JPA Specification.

- Classes in mapping files: the classes that have mapping entries, such as entity (see "entity" in the JPA Specification), mapped-superclass or embeddable, in an XML mapping file. For more information, see "mapped-superclass" and "embeddable" in the JPA Specification.

  If these classes are in the deployed component archive, then they will already be on the classpath. If they are not, you must explicitly include them in the classpath.

- Explicitly listed classes: the classes that are listed as class elements in the `persistence.xml` file. Consider listing classes explicitly if one of the following applies:
  - there are additional classes that are not local to the deployment unit JAR. For example, there is an embedded object class in a different JAR that you want to use in an entity in your persistence unit. You would list the fully qualified class in the class element in the `persitence.xml` file. You would also need to ensure that the JAR or directory that contains the class is on the classpath of the deployed component (by adding it to the manifest classpath of the deployment JAR, for example);
  - you want to exclude one or more classes that may be annotated as an entity. Even though the class may be annotated with the `@Entity` annotation, you do not want it treated as an entity in this particular deployed context. For example, you may want to use this entity as a transfer object and it needs to be part of the deployment unit. In this case, in the Jakarta EE environment, you have to use the `exclude-unlisted-classes` element of the `persistence.xml` file—the use of the default setting of this element prevents local classes from being added to the persistence unit. For more information, see "mapping-file, jar-file, class, exclude-unlisted-classes" of the JPA Specification.
  - you plan to run your application in the Java SE environment, and you list your classes explicitly because that is the only portable way to do so in Java SE.
- Additional JAR files of managed classes: the annotated classes in a named JAR file listed in a `jar-file` element in the `persistence.xml` file. For more information, see "mapping-file, jar-file, class, exclude-unlisted-classes" in the JPA Specification.

  You have to ensure that any JAR file listed in the `jar-file` element is on the classpath of the deployment unit. Do so by manually adding the JAR file to the manifest classpath of the

deployment unit.

Note that you must list the JAR file in the `jar-file` element relative to the parent of the JAR file in which the `persistence.xml` file is located. This matches what you would put in the classpath entry in the manifest file.

About the Deployment Classpath

To be accessible to the EJB JAR, WAR, or EAR file, a class or a JAR file must be on the deployment classpath. You can achieve this in one of the following ways:

- Put the JAR file in the manifest classpath of the EJB JAR or WAR file.

  To do this, add a classpath entry to the `META-INF/MANIFEST.MF` file in the JAR or WAR file. You can specify one or more directories or JAR files, separating them by spaces.

- Place the JAR file in the library directory of the EAR file.

  This will make the JAR file available on the application classpath and accessible by all of the modules deployed within the EAR file. By default, this would be the `lib` directory of the EAR file, although you may configure it to be any directory in the EAR file using the `library-directory` element in the `application.xml` deployment descriptor.

About Persistence Unit Packaging Options

Jakarta EE allows for persistence support in a variety of packaging configurations. You can deploy your application to the following module types:

- EJB modules: you can package your entities in an EJB JAR. When defining a persistence unit in an EJB JAR, the `persistence.xml` file is not optional–you must create and place it in the `META-INF` directory of the JAR alongside the deployment descriptor, if it exists.
- Web modules: you can use a WAR file to package your entities. In this case, place the `persistence.xml` file in the `WEB-INF/classes/META-INF` directory. Since the `WEB-INF/classes` directory is automatically on the classpath of the WAR, specify the mapping file relative to that directory.
- Persistence archives: a persistence archive is a JAR that contains a `persistence.xml` file in its `META-INF` directory and the managed classes for the persistence unit defined by the `persistence.xml` file. Use a persistence archive if you want to allow multiple components in different Jakarta EE modules to share or access a persistence unit.

  Once you create a persistence archive, you can place it in either the root or the application library directory of the EAR. Alternatively, you can place the persistence archive in the `WEB-INF/lib` directory of a WAR. This will make the persistence unit accessible only to the classes inside the WAR, but it enables the decoupling of the definition of the persistence unit from the web archive itself.

For more information, see "Persistence Unit Packaging" in the JPA Specification.

About the Persistence Unit Scope

You can define any number of persistence units in single `persistence.xml` file. The following are the rules for using defined and packaged persistence units:

- Persistence units are accessible only within the scope of their definition.
- Persistence units names must be unique within their scope.

For more information, see "Persistence Unit Scope" in the JPA Specification.

About Composite Persistence Units

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.

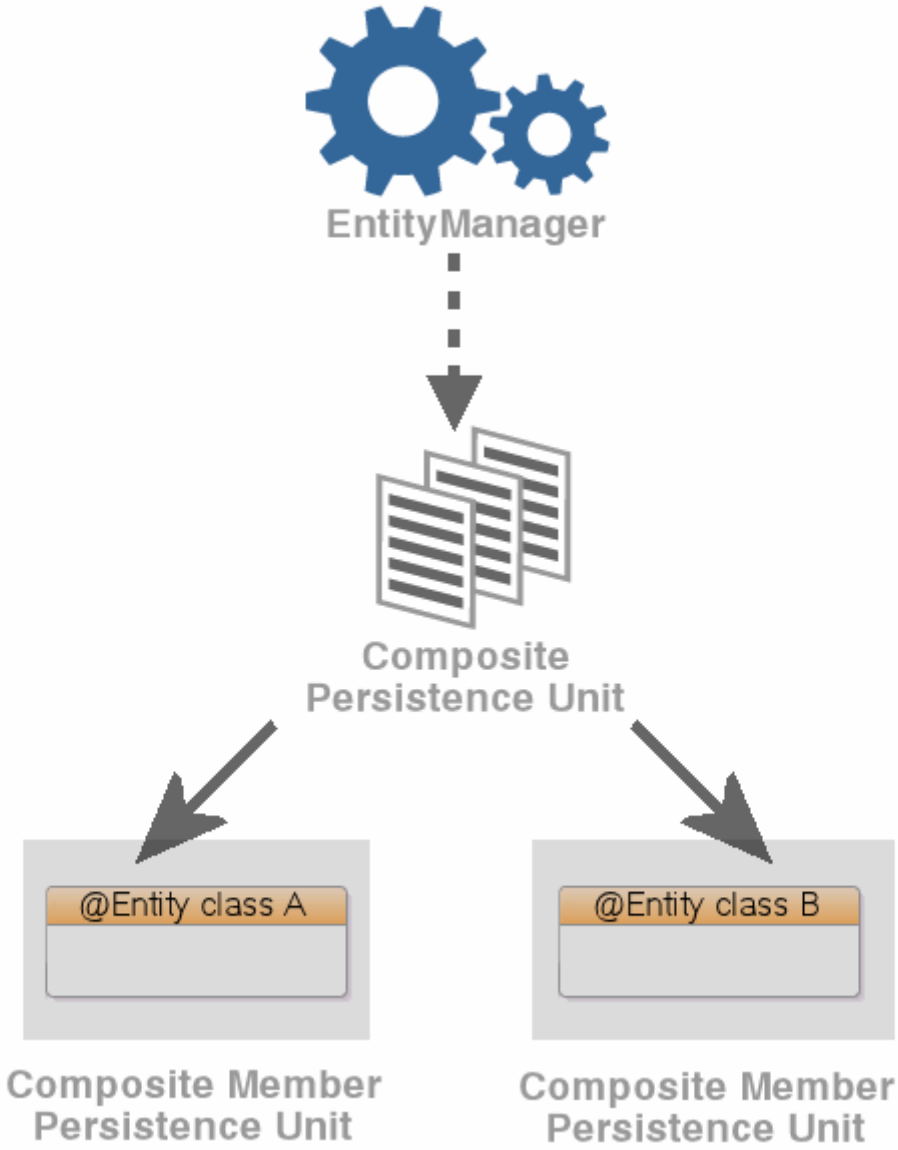
With a composite persistence unit, you can:

- 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

Figure 3-1 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 3-1 A Simple Composite Persistence Unit



Description of "Figure 3-1 A Simple Composite Persistence Unit"

For more information, see "Using Multiple Databases with a Composite Persistence Unit" in *Solutions Guide for EclipseLink*.

3.2. Building and Using the Persistence Layer

EclipseLink requires that classes must meet certain minimum requirements before they can become persistent. EclipseLink also provides alternatives to most requirements. EclipseLink uses a nonintrusive approach by employing a metadata architecture that allows for minimal object model intrusions.

This section includes the following information:

- Implementation Options
- Persistent Class Requirements
- Persistence Layer Components

Implementation Options

When implementing your persistence layer using EclipseLink, consider the following options:

- Using EclipseLink JPA Metatdata, Annotations, and XML
- Using EclipseLink Metadata Java API
- Using Method and Direct Field Access
- Using Java Byte-code Weaving

Using EclipseLink JPA Metatdata, Annotations, and XML

When using JPA, you can specify persistence layer components using any combination of standard JPA annotations and `persistence.xml`, EclipseLink JPA annotation extensions, and EclipseLink JPA `persistence.xml` extensions.

For more information, see About Configuration Basics.

Using EclipseLink Metadata Java API

Persistence layer components can be coded or generated as Java. To use Java code, you must manually write code for each element of the project including: project, login, platform, descriptors, and mappings. This may be more efficient if your application is model-based and relies heavily on code generation.

Using Method and Direct Field Access

You can access the fields (data members) of a class by using a getter/setter method (also known as property access) or by accessing the field itself directly.

When to use method or direct field access depends on your application design. Consider the following guidelines:

- Use method access outside of a class.

  This is the natural public API of the class. The getter/setter methods handle any necessary side-effects and the client need not know anything about those details.

- Use direct field access within a class to improve performance.

  In this case, you are responsible for taking into consideration any side-effects not invoked by bypassing the getter/setter methods.

When considering using method or direct field access, consider the following limitations.

If you enable change tracking on a getter/setter method (for example, you decorate method

`setPhone` with `@ChangeTracking`), then EclipseLink tracks changes accordingly when a client modifies the field (`phone`) using the getter/setter methods.

Similarly, if you enable change tracking on a field (for example, you decorate field `phone` with `@ChangeTracking`), then EclipseLink tracks changes accordingly when a client modifies the field (`phone`) directly.

However, if you enable change tracking on a getter/setter method (for example, you decorate method `setPhone` with `@ChangeTracking`) and a client accesses the field (`phone`) directly, EclipseLink does not detect the change. If you choose to code in this style of field access within a class for performance and method access outside of a class, then be aware of this limitation.

For more information, see the description of the `@ChangeTracking` annotation in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Using Java Byte-code Weaving

Weaving is a technique of manipulating the byte-code of compiled Java classes.

Weaving is used 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.

For more information, see About Weaving.

Persistent Class Requirements

When you create persistent Java objects, use direct access on private or protected attributes.

If you are using weaving, the `ValueHolderInterface` is not required. For more information, see About Weaving. See Indirection (Lazy Loading) for more information on indirection and transparent indirection.

Persistence Layer Components

The purpose of your application's persistence layer is to use a session at run time to associate mapping metadata and a data source (see Chapter 7, "Understanding Data Access") to create, read, update, and delete persistent objects using the EclipseLink cache, queries and expressions, and transactions.

Typically, the EclipseLink persistence layer contains the following components:

- Mapping Metadata
- Cache
- Queries and Expressions

Mapping Metadata

The EclipseLink application metadata model is based on the project. The project includes descriptors, mappings, and various policies that customize the run-time capabilities. You associate this mapping and configuration information with a particular data source and application by referencing the project from a session.

For more information, see the following:

- Creating Project Metadata
- Chapter 5, "Understanding Descriptors"
- Chapter 6, "Understanding Mappings"

Cache

By default, EclipseLink sessions provide an object-level cache that guarantees object identity and enhances performance by reducing the number of times the application needs to access the data source. EclipseLink provides a variety of cache options, including locking, refresh, invalidation, isolation, and coordination. Using cache coordination, you can configure EclipseLink to synchronize changes with other instances of the deployed application. You configure most cache options at the persistence unit or entity level. You can also configure cache options on a per-query basis or on a descriptor to apply to all queries on the reference class.

For more information, see Chapter 8, "Understanding Caching."

Queries and Expressions

For Object-relational architectures, EclipseLink provides several object and data query types, and offers flexible options for query selection criteria, including the following:

- EclipseLink expressions
- JPQL (Java Persistence Query Language)
- SQL
- Stored procedures
- Query by example

With these options, you can build any type of query. Oracle recommends using named queries to define application queries. Named queries are held in the project metadata and referenced by name. This simplifies application development and encapsulates the queries to reduce maintenance costs.

For Object-relational architectures, you are free to use any of the query options regardless of the

persistent entity type. Alternatively, you can build queries in code, using the EclipseLink API.

These query techniques cannot be used with MOXy (OXM, JAXB) mapping. However you can perform queries when using legacy EIS XML projects.

For more information, see Chapter 9, "Understanding Queries" and Chapter 10, "Understanding EclipseLink Expressions."

3.3. About Persisting Objects

This section includes a brief description of relational mapping and provides information and restrictions to guide object and relational modeling. This information is useful when building applications.

This section includes information on the following:

- Application Object Model
- Data Storage Schema
- Primary Keys and Object Identity
- Mappings
- Foreign Keys and Object Relationships
- Inheritance
- Concurrency
- Caching
- Nonintrusive Persistence
- Indirection
- Mutability

Application Object Model

Object modeling refers to the design of the Java classes that represent your application objects. You can use your favorite integrated development environment (IDE) or Unified Modeling Language (UML) modeling tool to define and create your application object model.

Any class that registers a descriptor with EclipseLink database sessions is called a persistent class. EclipseLink does not require that persistent classes provide public accessor methods for any private or protected attributes stored in the database. Refer to Persistent Class Requirements for more information.

Data Storage Schema

Your data storage schema refers to the design that you implement to organize the persistent data in your application. This schema refers to the data itself—not the actual data source (such as a relational database or nonrelational legacy system).

During the design phase of the application development process, you should decide how to implement the classes in the data source. When integrating existing data source information, you must determine how the classes relate to the existing data. If no legacy information exists to integrate, decide how you will store each class, then create the necessary schema.

Primary Keys and Object Identity

When making objects persistent, each object requires an *identity* to uniquely identify it for storage and retrieval. Object identity is typically implemented using a unique primary key. This key is used internally by EclipseLink to identify each object, and to create and manage references. Violating object identity can corrupt the object model.

In a Java application, object identity is preserved if each object in memory is represented by one, and only one, object instance. Multiple retrievals of the same object return references to the same object instance—not multiple copies of the same object.

EclipseLink supports multiple identity maps to maintain object identity (including composite primary keys). See About Cache Type and Size for additional information.

Mappings

EclipseLink uses metadata to describe how objects and beans map to the data source. This approach isolates persistence information from the object model—you are free to design their ideal object model, and DBAs are free to design their ideal schema. For more information, see About Metadata.

At run time, EclipseLink uses the metadata to seamlessly and dynamically interact with the data source, as required by the application.

EclipseLink provides an extensive mapping hierarchy that supports the wide variety of data types and references that an object model might contain. For more information, see Chapter 6, "Understanding Mappings."

Foreign Keys and Object Relationships

A **foreign key** can be one or more columns that reference a unique key, usually the primary key, in another table. Foreign keys can be any number of fields (similar to primary key), all of which are treated as a unit. A foreign key and the primary parent key it references must have the same

number and type of fields.

Foreign keys represents relationships from a column or columns in one table to a column or columns in another table. For example, if every `Employee` has an attribute `address` that contains an instance of `Address` (which has its own descriptor and table), the one-to-one mapping for the `address` attribute would specify foreign key information to find an address for a particular `Employee`.

Inheritance

Object-oriented systems allow classes to be defined in terms of other classes. For example: motorcycles, sedans, and vans are all *kinds of vehicles*. Each of the vehicle types is a *subclass* of the `Vehicle` class. Similarly, the `Vehicle` class is the *superclass* of each specific vehicle type. Each subclass inherits attributes and methods from its superclass (in addition to having its own attributes and methods).

Inheritance provides several application benefits, including the following:

- Using subclasses to provide specialized behaviors from the basis of common elements provided by the superclass. By using inheritance, you can reuse the code in the superclass many times.
- Implementing *abstract* superclasses that define generic behaviors. This abstract superclass may define and partially implement behavior, while allowing you to complete the details with specialized subclasses.

Concurrency

To have concurrent clients logged in at the same time, the server must spawn a dedicated thread of execution for each client. Jakarta EE application servers do this automatically. Dedicated threads enable each client to work without having to wait for the completion of other clients. EclipseLink ensures that these threads do not interfere with each other when they make changes to the identity map or perform database transactions. Your client can make transactional changes in an isolated and thread safe manner. EclipseLink manages clones for the objects you modify to isolate each client's work from other concurrent clients and threads. This is essentially an object-level transaction mechanism that maintains all of the ACID (Atomicity, Consistency, Isolation, Durability) transaction principles as a database transaction.

EclipseLink supports configurable optimistic and pessimistic locking strategies to let you customize the type of locking that the EclipseLink concurrency manager uses. For more information, see Descriptors and Locking.

Caching

EclipseLink caching improves application performance by automatically storing data returned as objects from the database for future use. This caching provides several advantages:

- Reusing Java objects that have been previously read from the database minimizes database access
- Minimizing SQL calls to the database when objects already exist in the cache
- Minimizing network access to the database
- Setting caching policies a class-by-class and bean-by-bean basis
- Basing caching options and behavior on Java garbage collection

EclipseLink supports several caching polices to provide extensive flexibility. You can fine-tune the cache for maximum performance, based on individual application performance. Refer to Chapter 8, "Understanding Caching" for more information.

Nonintrusive Persistence

The EclipseLink nonintrusive approach of achieving persistence through a metadata architecture means that there are almost no object model intrusions.

To persist Java objects, EclipseLink does not require any of the following:

- Persistent superclass or implementation of persistent interfaces
- Store, delete, or load methods required in the object model
- Special persistence methods
- Generating source code into or wrapping the object model

See Building and Using the Persistence Layer for additional information on this nonintrusive approach. See also About Metadata.

Indirection

An indirection object takes the place of an application object so the application object is not read from the database until it is needed. Using indirection, or lazy loading in JPA, allows EclipseLink to create *stand-ins* for related objects. This results in significant performance improvements, especially when the application requires the contents of only the retrieved object rather than all related objects.

Without indirection, each time the application retrieves a persistent object, it also retrieves *all* the objects referenced by that object. This may result in lower performance for some applications.

Oracle strongly recommends that you always use indirection.

EclipseLink provides several indirection models, such as proxy indirection, transparent indirection, and value holder indirection.

See Using Indirection with Collections and Indirection (Lazy Loading) for more information.

Mutability

Mutability is a property of a complex field that specifies whether the field value may be changed or not changed as opposed to replaced.

An immutable mapping is one in which the mapped object value cannot change unless the object ID of the object changes: that is, unless the object value is replaced by another object value altogether.

A mutable mapping is one in which the mapped object value can change without changing the object ID of the object.

By default, EclipseLink assumes the following:

- all `TransformationMapping` instances are mutable
- all JPA `@Basic` mapping types, except `Serializable` types, are immutable (including `Date` and `Calendar` types)
- all JPA `@Basic` mapping `Serializable` types are mutable

Whether a value is immutable or mutable largely depends on how your application uses your persistent classes. For example, by default, EclipseLink assumes that a persistent field of type `Date` is immutable: this means that as long as the value of the field has the same object ID, EclipseLink assumes that the value has not changed. If your application uses the set methods of the `Date` class, you can change the state of the `Date` object value without changing its object ID. This prevents EclipseLink from detecting the change. To avoid this, you can configure a mapping as mutable: this tells EclipseLink to examine the state of the persistent value, not just its object ID.

You can configure the mutability of the following:

- `TransformationMapping` instances;
- any JPA `@Basic` mapping type (including `Date` and `Calendar` types) individually;
- all `Date` and `Calendar` types.

Mutability can affect change tracking performance. For example, if a transformation mapping maps a mutable value, EclipseLink must clone and compare the value. If the mapping maps a simple immutable value, you can improve performance by configuring the mapping as immutable.

Mutability also affects weaving. EclipseLink can only weave an attribute change tracking policy for immutable mappings.

For more information, see About Weaving. See also the description of the `@Mutable` annotation in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

3.4. Migrating Applications to the EclipseLink Persistence Manager

You can configure an application server to use EclipseLink as the persistence manager. You can use

only one persistence manager for all of the entities with container-managed persistence in a JAR file.

EclipseLink provides automated support for migrating an existing Jakarta EE application to use EclipseLink as the persistence manager. For more information, see "Migrating from Apache OpenJPA to EclipseLink" in *Solutions Guide for EclipseLink*.

3.5. About 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, including inside a Jakarta EE 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.

This section describes the following:

- Using Dynamic Weaving
- Using Static Weaving
- Weaving POJO Classes
- Weaving and Jakarta EE Application Servers
- Disabling Weaving with Persistence Unit Properties

Using Dynamic Weaving

Use dynamic weaving to weave application class files one at a time, as they are loaded at run time. Consider this option when the number of classes to weave is few or when the time taken to weave the classes is short.

If the number of classes to weave is large or the time required to weave the classes is long, consider using static weaving.

Using Static Weaving

Use static weaving to weave all application class files at build time so that you can deliver prewoven class files. Consider this option to weave all applicable class files at build time so that you can deliver prewoven class files. By doing so, you can improve application performance by eliminating the runtime weaving step required by dynamic weaving.

In addition, consider using static weaving to weave in Java environments where you cannot

configure an agent.

Weaving POJO Classes

EclipseLink uses weaving to enable the following for POJO classes:

- Lazy loading
- Change tracking
- Fetch groups

EclipseLink weaves all the POJO classes in the JAR you create when you package a POJO application for weaving.

EclipseLink weaves all the classes defined in the `persistence.xml` file, that is:

- All the classes you list in the `persistence.xml` file.
- All classes relative to the JAR containing the `persistence.xml` file if element `<exclude-unlisted-classes>` is false.

Weaving and Jakarta EE Application Servers

The default EclipseLink weaving behavior applies in any Jakarta EE JPA-compliant application server using the EclipseLink JPA persistence provider. To change this behavior, modify your `persistence.xml` file (for your JPA entities or POJO classes) to use EclipseLink JPA properties, EclipseLink JPA annotations, or both.

Disabling Weaving with Persistence Unit Properties

To disable weaving using EclipseLink persistence unit properties, configure your `persistence.xml` file with one or more of the following properties set to false:

- `eclipse.weaving`; disables all weaving
- `eclipselink.weaving.lazy`; disables weaving for lazy loading (indirection)
- `eclipselink.weaving.changetracking`; disables weaving for change tracking
- `eclipselink.weaving.fetchgroups`; disables weaving for fetch groups
- `eclipselink.weaving.internal`; disables weaving for internal optimization
- `eclipselink.weaving.eager`; disables weaving for indirection on eager relationships

Chapter 4. Understanding Entities

This chapter introduces and describes entities. An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in the table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

You can configure your entity's identity, as well as the locking technique and sequence generation options for your entity.

This chapter includes the following sections:

4.1. Identifying Entities

Use the `@Entity` annotation to specify that a class is an entity.

The entity class must also be listed in your `persistence.xml` file, unless you set the tag `<exclude-unlisted-classes>` to false.

For more information, see Chapter 2 "Entities" in the JPA Specification.

`http://jcp.org/en/jsr/detail?id=338`

4.2. Entities and Persistent Identity

Every entity must have a persistent identity, which is an equivalent of a primary key in a database table that stores the entity state.

By default, the EclipseLink persistence provider assumes that each entity has at least one field or property that serves as a primary key.

You can generate and/or configure the identity of your entities by using the following annotations:

- `@Id`
- `@IdClass`
- `@EmbeddedId`
- `@GeneratedValue`
- `@TableGenerator`
- `@SequenceGenerator`
- `@UuidGenerator`

You can also use these annotations to fine-tune how your database maintains the identity of your entities. For more information on these annotations, see "Metadata for Object/Relational Mapping" in the JPA Specification.

`http://jcp.org/en/jsr/detail?id=338`

4.3. Entities and Database Tables

Every entity class maps to a specific table or set of tables in the database. By default, the entity's table name is defaulted as its entity name as uppercase, which defaults to the entity's short class name. An entity normally maps to a single table, but can also map to multiple tables, or even a view.

You can customize an entity's tables using the following annotations:

- `@Table`
- `@SecondaryTable`

4.4. Entities and Inheritance

JPA defines several difference methods for persisting objects with inheritance. The `@Inheritance` annotation is used in the root class to define `SINGLE_TABLE, JOINED`, and `TABLE_PER_CLASS` inheritance. For abstract classes that define common state or persistence behavior, but have no relationship on the database, the `@MappedSuperclass` annotation can be used.

- `@Inheritance`
- `@MappedSuperclass`

4.5. Entities and Embedded Objects

You can use the `@Embeddable` annotation to map an embedded class. An embeddable is a special type of class that is not directly persistent, but persisted only with its parent entity. An embeddable can be referenced from an entity or another embeddable using the `@Embedded` annotation for a single reference, `@EmbeddedId` for an embedded id, or the `@ElementCollection` annotation for a `Collection` or `Map` reference. An embeddable can also be used in any `Map` key using the `@MapKeyClass` annotation.

- `@Embeddable`

- `@EmbeddedId`
- `@Embedded`
- `@ElementCollection`

4.6. Entities and Sequence Generation

Many databases support an internal mechanism for id generation called sequences. You can use a database sequence to generate identifiers when the underlying database supports them.

- `@SequenceGenerator`—If you use the `@GeneratedValue` annotation to specify a primary key generator of type `SEQUENCE`, then you can use the `@SequenceGenerator` annotation to fine-tune this primary key generator to do the following:
  - change the allocation size to match your application requirements or database performance parameters
  - change the initial value to match an existing data model (for example, if you are building on an existing data set for which a range of primary key values has already been assigned or reserved)
  - use a predefined sequence in an existing data model
- `@TableGenerator`—If you use the `@GeneratedValue` annotation to specify a primary key generator of type `TABLE`, then you can use the `@TableGenerator` annotation to fine-tune this primary key generator to do the following:
  - change the name of the primary key generator's table, because the name is awkward, a reserved word, incompatible with a preexisting data model, or invalid as a table name in your database
  - change the allocation size to match your application requirements or database performance parameters
  - change the initial value to match an existing data model (for example, if you are building on an existing data set, for which a range of primary key values has already been assigned or reserved)
  - configure the primary key generator's table with a specific catalog or schema
  - configure a unique constraint on one or more columns of the primary key generator's table

For more information and examples of these annotations, see "Metadata for Object/Relational Mapping" in the JPA Specification.

4.7. Entities and Locking

You have the choice between optimistic and pessimistic locking. Oracle recommends using optimistic locking. For more information, see Descriptors and Locking.

By default, the EclipseLink persistence provider assumes that the application is responsible for data consistency.

Oracle recommends that you use the `@Version` annotation to enable JPA-managed optimistic locking

by specifying the version field or property of an entity class that serves as its optimistic lock value. When choosing a version field or property, ensure that the following is true:

- there is only one version field or property per entity
- you choose a property or field persisted to the primary table (see "Table Annotation" in the JPA Specification)
- your application does not modify the version property or field

The field or property type must either be a numeric type (such as `Number`, `long`, `int`, `BigDecimal`, and so on), or a `java.sql.Timestamp`. Oracle recommends using a numeric type.

The `@Version` annotation does not have attributes.

For more information, see the following:

- "Optimistic Locking and Concurrency" in the JPA Specification `http://jcp.org/en/jsr/detail?id=338`
- "Version Annotation" in the JPA Specification `http://jcp.org/en/jsr/detail?id=338`
- EclipseLink JPA extensions for optimistic locking described in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*

For more information on the EclipseLink artifacts configured by JPA metadata, see Descriptors and Locking.

4.8. Extensible Entities

JPA entities and JAXB beans can be made extensible by adding or modifying mappings externally. There is no need to modify the entity or bean source file nor do you have to redeploy the 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.

Use the `@VirtualAccessMethods` annotation to specify that a JPA entity is extensible and use the `@XmlVirtualAccessMethods` annotation to specify that a JAXB bean is extensible. In both cases, you use virtual properties to specify external mappings. This allows you to modify the mappings

without modifying source files and without redeploying the persistence unit.

For information on how to make JPA entities and JAXB beans extensible, see "Making JPA Entities and JAXB Beans Extensible" in *Solutions Guide for EclipseLink*.

Chapter 5. Understanding Descriptors

This chapter introduces and describes descriptors. EclipseLink uses descriptors to store the information that describes how an instance of a particular class can be represented by a data source. Descriptors own mappings that associate class instance variables with a data source and transformation routines that are used to store and retrieve values. As such, the descriptor acts as the connection between a Java object and its data source representation.

This chapter includes the following sections:

5.1. Common Descriptor Concepts

The following sections describe the concepts that are common to Object-Relational and MOXy descriptors.

Descriptor Architecture

A **descriptor** stores all the information describing how an instance of a particular object class can be represented in a data source. The Descriptor API can be used to define, or amend EclipseLink descriptors through Java code. The Descriptor API classes are mainly in the `org.eclipse.persistence.descriptors` package.

EclipseLink descriptors may contain the following information:

- The persistent Java class it describes and the corresponding data source (database tables or XML complex type interaction)
- A collection of mappings, which describe how the attributes and relationships for that class are represented in the data source
- The primary key information (or equivalent) of the data source
- A list of query keys (or aliases) for field names
- Information for sequence numbers

- A set of optional properties for tailoring the behavior of the descriptor, including support for caching refresh options, identity maps, optimistic locking, the event manager, and the query manager

There is a descriptor type for each data source type that EclipseLink supports. In some cases, multiple descriptor types are valid for the same data source type. The type of descriptor you use determines the type of mappings that you can define.

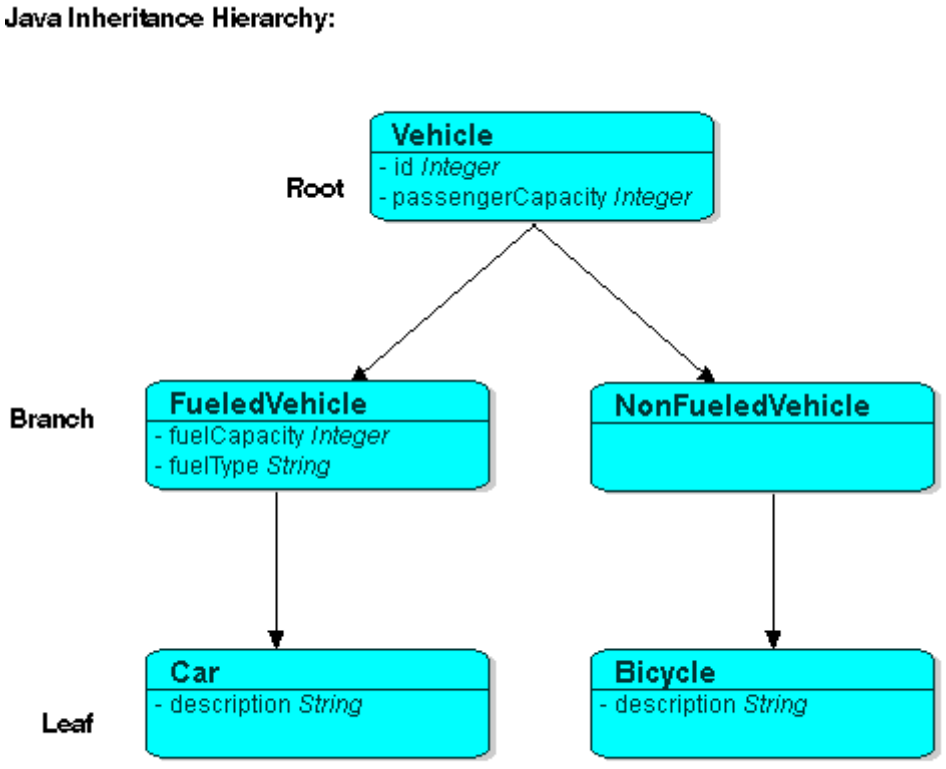
Descriptors and Inheritance

Inheritance describes how a derived (child) class inherits the characteristics of its superclass (parent). You can use descriptors to describe the inheritance relationships between classes in relational and XML projects.

In the descriptor for a child class, you can override mappings that have been specified in the descriptor for a parent class, or map attributes that have not been mapped at all in the parent class descriptor.

Figure 5-1 illustrates the `Vehicle` object model–a typical Java inheritance hierarchy. The root class `Vehicle` contains two branch classes: `FueledVehicle` and `NonFueledVehicle`. Each branch class contains a leaf class: `Car` and `Bicycle`, respectively.

Figure 5-1 Example Inheritance Hierarchy



Description of "Figure 5-1 Example Inheritance Hierarchy "

EclipseLink recognizes the following three types of classes in an inheritance hierarchy:

- The root class stores information about *all* instantiable classes in its subclass hierarchy. By default, queries performed on the root class return instances of the root class and its instantiable subclasses. However, the root class can be configured so queries on it return only instances of itself, without instances of its subclasses.

  For example, the `Vehicle` class in Figure 5-1 is a root class.

- Branch classes have a persistent superclass and also have subclasses. By default, queries performed on the branch class return instances of the branch class and any of its subclasses. However, as with the root class, the branch class can be configured so queries on it return only instances of itself without instances of its subclasses.

  For example, the `FueledVehicle` class in Figure 5-1 is a branch class.

- Leaf classes have a persistent superclass in the hierarchy but do not have subclasses. Queries performed on the leaf class can only return instances of the leaf class.

  For example, the `Car` class in Figure 5-1 is a leaf class.

In the descriptor for a child class, you can override mappings that have been specified in the descriptor for a parent class, or map attributes that have not been mapped at all in the parent class descriptor.

This section includes information on the following topics:

- Specifying a Class Indicator
- Inheritance and Primary Keys
- Single and Multi-Table Inheritance
- Aggregate and Composite Descriptors and Inheritance

Specifying a Class Indicator

When configuring inheritance, you configure the root class descriptor with the means of determining which subclasses it should instantiate.

You can do this in one of the following ways:

- Using Class Indicator Fields
- Using Class Extraction Methods

All leaf classes in the hierarchy must have a class indicator and they must have the same type of class indicator (field or class extraction method).

Using Class Indicator Fields

You can use a persistent attribute of a class to indicate which subclass should be instantiated. For example, in a relational descriptor, you can use a class indicator field in the root class table. The indicator field should not have an associated direct mapping unless it is set to read-only.

If the indicator field is part of the primary key, define a write-only transformation mapping for the indicator field.

You can use strings or numbers as values in the class indicator field.

The root class descriptor must specify how the value in the class indicator field translates into the class to be instantiated.

Using Class Extraction Methods

You can define a Java method to compute the class indicator based on any available information in the object's data source record. Such a method is called a class extraction method.

Using a class extraction method, you do not need to include an explicit class indicator field in your data model and you can handle relationships that are too complex to describe using class indicator fields.

A class extraction method must have the following characteristics:

- it must be defined on the root descriptor's class;
- it must be static;
- it must take a `Record` as an argument;
- it must return the `java.lang.Class` object to use for the `Record` passed in.

You may also need to define only-instances and with-all-subclasses expressions. If you use a class extraction method, then you must provide EclipseLink with expressions to correctly filter sibling instances for all classes that share a common table.

When configuring inheritance using a class extraction method, EclipseLink does not generate SQL for queries on the root class.

Inheritance and Primary Keys

For relational projects, EclipseLink assumes that all of the classes in an inheritance hierarchy have the same primary key, as set in the root descriptor.

Single and Multi-Table Inheritance

In a relational project, you can map your inheritance hierarchy to a single table or to multiple tables.

Aggregate and Composite Descriptors and Inheritance

You can designate relational descriptors as aggregates. XML descriptors are always composites (see Descriptors and Aggregation).

When configuring inheritance for a relational aggregate descriptor, all the descriptors in the inheritance tree must be aggregates. The descriptors for aggregate and non-aggregate classes cannot exist in the same inheritance tree.

When configuring inheritance for an XML descriptor, because all XML descriptors are composites, descriptor type does not restrict inheritance.

Descriptors and Aggregation

Two objects—a source (parent or owning) object and a target (child or owned) object—are related by aggregation if there is a strict one-to-one relationship between them, and all the attributes of the target object can be retrieved from the same data source representation as the source object. This means that if the source object exists, then the target object must also exist, and if the source object is destroyed, then the target object is also destroyed.

In this case, the descriptors for the source and target objects must be designated to reflect this relationship.

The EJB 3.0 specification does not support nested aggregates).

Descriptor Customization

You can customize a descriptor at run time by specifying a descriptor customizer—a Java class that implements the `org.eclipse.persistence.config.DescriptorCustomizer` interface and provides a default (zero-argument) constructor.

You use a descriptor customizer to customize a descriptor at run time through code API similar to how you use an amendment method to customize a descriptor. See Amendment Methods.

Amendment Methods

You can associate a static Java method that is called when a descriptor is loaded at run time. This

method can amend the run-time descriptor instance through the descriptor Java code API. The method must be `public` `static` and take a single parameter of type `org.persistence.descriptors.structures.ClassDescriptor`. In the implementation of this method, you can configure advanced features of the descriptor using any of the public descriptor and mapping API.

You can only modify descriptors before the session has been connected; you should not modify descriptors after the session has been connected.

Amendment methods can be used with rational descriptors, object-relational data type descriptors, and XML descriptors.

Descriptor Event Manager

In relational projects, EclipseLink raises various instances of `DescriptorEvent` during the persistence life cycle. Each descriptor owns an instance of `DescriptorEventManager` that is responsible for receiving these events and dispatching them to the descriptor event handlers registered with it.

Using a descriptor event handler, you can execute your own application specific logic whenever descriptor events occur, allowing you to take customized action at various points in the persistence life-cycle. For example, using a descriptor event handler, you can do the following:

- Synchronize persistent objects with other systems, services, and frameworks
- Maintain nonpersistent attributes of which EclipseLink is not aware
- Notify other objects in the application when the persistent state of an object changes
- Implement complex mappings or optimizations not directly supported by EclipseLink mappings

5.2. Object-Relational Descriptor Concepts

The following sections describe the concepts specific to Object-Relational descriptors.

Fetch Groups

By default, when you execute an object-level read query for a particular object class, EclipseLink returns all the persistent attributes mapped in the object's descriptor. With this single query, all the object's persistent attributes are defined, and calling their `get` methods returns the value directly from the object.

When you are interested in only some of the attributes of an object, it may be more efficient to return only a subset of the object's attributes using a fetch group.

Using a fetch group, you can define a subset of an object's attributes and associate the fetch group with either a `ReadObjectQuery` or `ReadAllQuery` query. When you execute the query, EclipseLink retrieves only the attributes in the fetch group. EclipseLink automatically executes a query to fetch all the attributes excluded from this subset when and if you call a get method on any one of the excluded attributes.

You can define more than one fetch group for a class. You can optionally designate at most one such fetch group as the default fetch group. If you execute either a `ReadObjectQuery` or `ReadAllQuery` query without specifying a fetch group, EclipseLink will use the default fetch group, unless you configure the query otherwise.

Before using fetch groups, Oracle recommends that you perform a careful analysis of system use. In many cases, the extra queries required to load attributes not in the fetch group could well offset the gain from the partial attribute loading.

Fetch groups can be used only with basic mappings configured with `FetchType.LAZY (partial object queries)`.

EclipseLink uses the `AttributeGroup` that can be used to configure the use of partial entities in fetch, load, copy, and merge operations.

- Fetch: Control which attributes and their associated columns are retrieved from the database
- Load: Control which relationships in the entities returned from a query are populated
- Copy: Control which attributes are copied into a new entity instance
- Merge: Merge only those attributes fetched, loaded, or copied into an entity

AttributeGroup Types and Operations

The following sections describe the possible `AttributeGroup` types and operations.

- FetchGroup
- Default FetchGroup
- Named FetchGroup
- Full FetchGroup
- Load/LoadAll with FetchGroup
- LoadGroup
- CopyGroup
- Merging

FetchGroup

The `FetchGroup` defines which attributes should be fetched (selected from the database) when the entity is retrieved as the result of a query execution. The inclusion of relationship attributes in a

`FetchGroup` only determines if the attribute's required columns should be fetched and populated. In the case of a lazy fetch type the inclusion of the attribute simply means that its proxy will be created to enable lazy loading when accessed. To force a relationship mapping to be populated when using a `FetchGroup` on a query the attribute must be included in the group and must either be `FetchType.EAGER` or it must be included in an associated `LoadGroup` on the query.

Default FetchGroup

`FetchGroup` also has the notion of named and default `FetchGroup` which are managed by the `FetchGroupManager`. A default `FetchGroup` is defined during metadata processing if one or more basic mappings are configured to be lazy and the entity class implements `FetchGroupTracker` (typically introduced through weaving). The default `FetchGroup` is used on all queries for this entity type where no explicit `FetchGroup` or named `FetchGroup` is configured.

Named FetchGroup

A named `FetchGroup` can be defined for an entity using `@FetchGroup` annotation or within the `eclipselink-orm.xml` file.

Full FetchGroup

A `FetchGroup` when first created is assumed to be empty. The user must add the attributes to the `FetchGroup`. If a `FetchGroup` is required with all of the attributes then the `FetchGroupManager.createFullFetchGroup()` must be used.

Load/LoadAll with FetchGroup

A `FetchGroup` can also be configured to perform a load operation of relationship mappings and nested relationship mappings.

LoadGroup

A `LoadGroup` is used to force a specified set of relationship attributes to be populated in a query result.

CopyGroup

The `CopyGroup` replaces the deprecated `ObjectCopyPolicy` being used to define how a entity is copied. In addition to specifying the attributes defining what should be copied from the source entity graph into the target copy the `CopyGroup` also allows definition of:

- `shouldResetPrimaryKey`: Reset the identifier attributes to their default value. This is used when the copy operation is intended to clone the entity in order to make a new entity with similar state to the source. Default is `false`.
- `shouldResetVersion`: Reset the optimistic version locking attribute to its default value in the copies. Default is `false`.
- `depth`: defines cascade mode for handling relationships. By default `CASCADE_PRIVATE_PARTS` is used but it can also be configured to `NO_CASCADE` and `CASCADE_ALL_PARTS`.

Merging

When a partial entity is merged into a persistence context that has an `AttributeGroup` associated with it defining which attributes are available only those attributes are merged. The relationship mappings within the entity are still merged according to their cascade merge settings.

Descriptor Query Manager

Each relational descriptor provides an instance of `DescriptorQueryManager` that you can use to configure the following:

- named queries
- custom default queries for basic persistence operations
- additional join expressions

Descriptors and Sequencing

An essential part of maintaining object identity is managing the assignment of unique values (that is, a specific sequence) to distinguish one object instance from another.

Sequencing options you configure at the project (or session) level determine the type of sequencing that EclipseLink uses. In a POJO project, you can use session-level sequence configuration to override project-level sequence configuration, on a session-by-session basis, if required.

After configuring the sequence type, for each descriptor's reference class, you must associate one attribute, typically the attribute used as the primary key, with its own sequence.

Descriptors and Locking

With object-relational mapping, you can configure a descriptor with any of the following locking policies to control concurrent access to a domain object:

- Optimistic—All users have read access to the data. When a user attempts to make a change, the application checks to ensure the data has not changed since the user read the data.
- Pessimistic—The first user who accesses the data with the purpose of updating it locks the data until completing the update.
- No locking—The application does not prevent users overwriting each other's changes.

Oracle recommends using optimistic locking for most types of applications to ensure that users do not overwrite each other's changes.

This section describes the various types of locking policies that EclipseLink supports, including the following:

- Optimistic Version Locking Policies
- Pessimistic Locking Policies
- Applying Locking in an Application

Optimistic Version Locking Policies

With optimistic locking, all users have read access to the data. When a user attempts to make a change, the application checks to ensure the data has not changed since the user read the data.

Optimistic version locking policies enforce optimistic locking by using a version field (also known as a write-lock field) that you provide in the reference class that EclipseLink updates each time an object change is committed.

EclipseLink caches the value of this version field as it reads an object from the data source. When the client attempts to write the object, EclipseLink compares the cached version value with the current version value in the data source in the following way:

- If the values are the same, EclipseLink updates the version field in the object and commits the changes to the data source.
- If the values are different, the write operation is disallowed because another client must have updated the object since this client initially read it.

EclipseLink provides the following version-based optimistic locking policies:

- `VersionLockingPolicy`
- `TimestampLockingPolicy`

For descriptions of these locking policies, see "Setting Optimistic Locking" in *Solutions Guide for EclipseLink*.

In general, Oracle recommends numeric version locking because of the following:

- accessing the timestamp from the data source can have a negative impact on performance;
- time stamp locking is limited to the precision that the database stores for timestamps.

Whenever any update fails because optimistic locking has been violated, EclipseLink throws an `OptimisticLockException`. This should be handled by the application when performing any database modification. The application must notify the client of the locking contention, refresh the object, and have the client reapply its changes.

You can choose to store the version value in the object as a mapped attribute, or in the cache. In three-tier applications, you typically store the version value in the object to ensure it is passed to the client when updated (see Applying Locking in an Application).

If you store the version value in the cache, you do not need to map it. If you do map the version field, you must configure the mapping as read-only.

To ensure that the parent object's version field is updated whenever a privately owned child object is modified, consider Optimistic Version Locking Policies and Cascading.

If you are using a stored procedure to update or delete an object, your database may not return the row-count required to detect an optimistic lock failure, so your stored procedure is responsible for checking the optimistic lock version and throwing an error if they do not match. Only version locking is directly supported with a `StoredProcedureCall`. Because timestamp and field locking require two versions of the same field to be passed to the call, an SQL call that uses an `##` parameter to access the translation row could be used for other locking policies.

Optimistic Version Locking Policies and Cascading

If your database schema is such that both a parent object and its privately owned child object are stored in the same table, then if you update the child object, the parent object's version field will be updated.

However, if the parent and its privately owned child are stored in separate tables, then changing the child will not, by default, update the parent's version field.

To ensure that the parent object's version field is updated in this case, you can either manually update the parent object's version field or, if you are using a `VersionLockingPolicy`, you can configure EclipseLink to automatically cascade the child object's version field update to the parent.

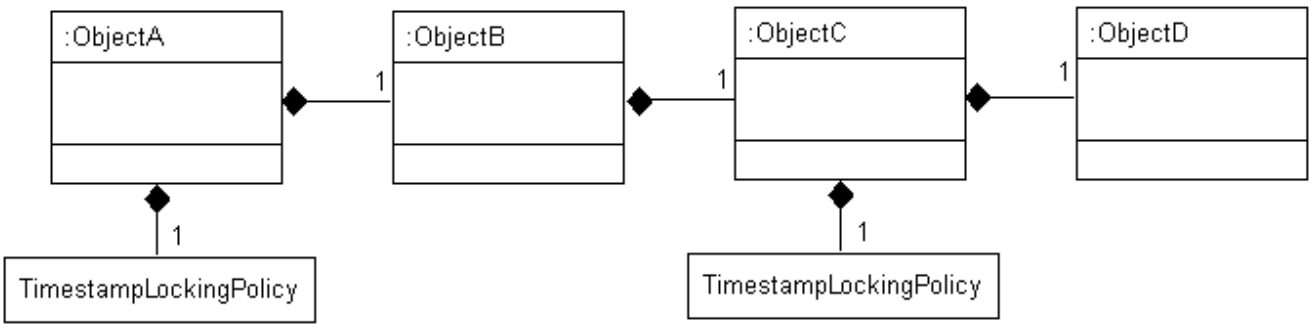
After you enable optimistic version locking cascading, when a privately owned child object is modified, EclipseLink will traverse the privately owned foreign reference mappings, updating all the parent objects back to the root.

EclipseLink supports optimistic version locking cascading for:

- object changes in privately owned one-to-one and one-to-many mappings
- relationship changes (adding or removing) in the following collection mappings (privately owned or not):
  - direct collection
  - one-to-many
  - many-to-many
  - aggregate collection

Consider the example object graph shown in Figure 5-2

Figure 5-2 Optimistic Version Locking Policies and Cascading Example



Description of "Figure 5-2 Optimistic Version Locking Policies and Cascading Example"

In this example, `ObjectA` privately owns `ObjectB`, and `ObjectB` privately owns `ObjectC`, and `ObjectC` privately owns `ObjectD`.

Suppose you register `ObjectB` in a unit of work, modify an `ObjectB` field, and commit the unit of work. In this case, `ObjectB` checks the cache for `ObjectA` and, if not present, queries the database for `ObjectA`. `ObjectB` then notifies `ObjectA` of its change. `ObjectA` forces an update on its version optimistic locking field even though it has no changes to its corresponding table.

Suppose you register `ObjectA` in a unit of work, access its `ObjectB` to access its `ObjectC` to access its `ObjectD`, modify an `ObjectD` field, and commit the unit of work. In this case, `ObjectD` notifies `ObjectC` of its changes. `ObjectC` forces an update on its version optimistic locking field even though it has no changes to its corresponding table. `ObjectC` then notifies `ObjectB` of the `ObjectD` change. `ObjectB` then notifies `ObjectA` of the `ObjectD` change. `ObjectA` forces an update on its version optimistic locking field even though it has no changes to its corresponding table.

Optimistic Locking and Rollbacks

With optimistic locking, use the `UnitOfWork` method `commitAndResumeOnFailure` to rollback a locked object's value, if you store the optimistic lock versions in the cache.

If you store the locked versions in an object, you must refresh the objects (or their versions) on a failure. Alternatively, you can acquire a new unit of work on the failure and reapply any changes into the new unit of work.

Optimistic Field Locking Policies

Optimistic field locking policies enforce optimistic locking by using one or more of the fields that currently exist in the table to determine if the object has changed since the client read the object.

The unit of work caches the original state of the object when you first read the object or register it with the unit of work. At commit time, the unit of work compares the original values of the lock fields with their current values on the data source during the update. If any of the lock fields' values have changed, an optimistic lock exception is thrown.

EclipseLink provides the following optimistic field locking policies:

- `AllFieldsLockingPolicy`
- `ChangedFieldsLockingPolicy`
- `SelectedFieldsLockingPolicy`
- `VersionLockingPolicy`
- `TimestampLockingPolicy`

For descriptions of these locking policies, see "Setting Optimistic Locking" in *Solutions Guide for EclipseLink*.

Pessimistic Locking Policies

With pessimistic locking, the first user who accesses the data with the purpose of updating it locks the data until completing the update.

When using a pessimistic locking policy, you can configure the policy to either fail immediately or to wait until the read lock is acquired.

You can use a pessimistic locking policy only in a project with a container-managed persistence type and with descriptors that have EJB information.

You can also use pessimistic locking (but not a pessimistic locking policy) at the query level.

EclipseLink provides an optimization for pessimistic locking when this locking is used with entities with container-managed persistence: if you set your query to pessimistic locking and run the query in its own new transaction (which will end after the execution of the finder), then EclipseLink overrides the locking setting and does not append `FOR UPDATE` to the SQL. However, the use of this optimization may produce an undesirable result if the pessimistic lock query has been customized by the user with a SQL string that includes `FOR UPDATE`. In this case, if the conditions for the optimization are present, the query will be reset to nonpessimistic locking, but the SQL will remain the same resulting in the locking setting of the query conflicting with the query's SQL string. To avoid this problem, you can take one of the following two approaches:

- Use an expression (see Chapter 10, "Understanding EclipseLink Expressions") for the selection criteria. This will give EclipseLink control over the SQL generation.
- Place the finder in a transaction to eliminate conditions for the optimization.

Applying Locking in an Application

To correctly lock an object in an application, you must obtain the lock before the object is sent to the client for editing.

Applying Optimistic Locking in an Application

If you are using optimistic locking, you have the following two choices for locking objects correctly:

- Map the optimistic lock field in your object as not read-only and pass the version to the client on the read and back to the server on the update.

  Ensure that the original version value is sent to the client when it reads the object for the update. The client must then pass the original version value back with the update information, and this version must be set into the object to be updated after it is registered/read in the new unit of work on the server.

- Hold the unit of work for the duration of the interaction with the client.

  Either through a stateful session bean, or in an HTTP session, store the unit of work used to read the object for the update for the duration of the client interaction.

  You must read the object through this unit of work before passing it to the client for the update. This ensures that the version value stored in the unit of work cache or in the unit of work clone will be the original value.

  This same unit of work must be used for the update.

The first option is more commonly used, and is required if developing a stateless application.

Applying Pessimistic Locking in an Application

If you are using pessimistic locking, you must use the unit of work to start a database transaction before the object is read. You must hold this unit of work and database transaction while the client is editing the object and until the client updates the object. You must use this same unit of work to update the object.

5.3. Descriptor Files

The following sections describe the descriptor files that can be used for object-relational and MOXy mapping.

- Using orm.xml for Object-Relational Mappings
- Using eclipselink-orm.xml for EclipseLink Object-Relational Mappings
- Using eclipselink-oxm.xml for EclipseLink MOXy Mappings

Using orm.xml for Object-Relational Mappings

Use the `orm.xml` file to apply the metadata to the persistence unit. This metadata is a union of all the mapping files and the annotations (if there is no `xml-mapping-metadata-complete` element). If you use one mapping `orm.xml` file for your metadata and place this file in a `META-INF` directory on the classpath, then you do not need to explicitly list it. The persistence provider will automatically search for this file (`orm.xml`) and use it.

The schema for the JPA 2.0 `orm.xml` is `orm_2_0.xsd`. (`http://java.sun.com/xml/ns/persistence/orm_2_0.xsd`)

If you use a different name for your mapping files or place them in a different location, you must list them in the `mapping-file` element of the `persistence.xml` file.

Using eclipselink-orm.xml for EclipseLink Object-Relational Mappings

EclipseLink supports an extended JPA `orm.xml` mapping configuration file called `eclipselink-orm.xml`. This mapping file can be used in place of JPA's standard mapping file or can be used to override a JPA mapping file. In additional to allowing all of the standard JPA mapping capabilities it also includes advanced mapping types and options.

For more information on the `eclipselink-orm.xml` file, see "eclipselink-orm.xml Schema Reference" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Using this mapping file enables many EclipseLink advanced features, but it may prevent the persistence unit from being portable to other JPA implementations.

For more information, on overriding values, see:

- "XML Overriding Rules" in the JPA Specification.

  `http://jcp.org/en/jsr/detail?id=338`

- The schema for `eclipselink-orm.xml` is `eclipselink_orm_2_2.xsd`:

  `http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_2.xsd`

Overriding and Merging Mapping Information

To override the `orm.xml` file's mapping, you must define the `META-INF/eclipselink-orm.xml` file in the project. The contents of `eclipselink-orm.xml` override `orm.xml` and any other JPA mapping file specified in the persistence unit. If there are overlapping specifications in multiple ORM files, the files are merged if there are no conflicting entities.

For more information, see "Overriding and Merging" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Validating the XML Schema

By default the content of your `.orm` XML file is not validated against the JPA `.orm` XML schema.

During development it is a good idea to validate your `.orm` XML file against the schema to ensure it is valid. In EclipseLink, validating the `.orm` XML schema can be enabled using the persistence unit property "eclipselink.orm.validate.schema" in your `persistence.xml` file.

Advantages and Disadvantages of Using XML

Some advantages of using XML instead of annotations include:

- No coupling between the metadata and the source code
- Compliance with the existing, pre-EJB 3.0 development process
- Support in IDEs and source control systems

The main disadvantages of mapping with XML include:

- It is inherently complex (when compared to annotations)
- The need for replication of the code context (that is, defining the structure in both the XML and the source code)

For more information, see Chapter 10 "Metadata Annotations" in the JPA Specification:

`http://jcp.org/en/jsr/detail?id=338`

Using eclipselink-oxm.xml for EclipseLink MOXy Mappings

You can use Java annotations to specify JAXB features in your projects. In addition to Java annotations, EclipseLink provides an XML mapping configuration file called `eclipselink-oxm.xml`. This mapping file contains the standard JAXB mappings and configuration options for advanced mapping types. You can use the `eclipselink-oxm.xml` file in place of or to override JAXB annotations in source code.

Using this mapping file will enable many advanced features but it can prevent the model from being portable to other JAXB implementations.

Chapter 6. Understanding Mappings

This chapter introduces and describes mappings. EclipseLink can transform data between an object representation and a representation specific to a data source. This transformation is called mapping and it is the core of EclipseLink projects.

A mapping corresponds to a single data member of a domain object. It associates the object data member with its data source representation and defines the means of performing the two-way conversion between object and data source.

This chapter includes the following sections:

- Common Mapping Concepts
- Object-Relational Mapping Concepts
- MOXy Mapping Concepts
- Object-JSON Mapping Concepts

6.1. Common Mapping Concepts

This section describes concepts for relational and nonrelational mappings that are unique to EclipseLink:

- Mapping Architecture
- Mapping Examples
- Mapping Converters
  - Serialized Object Converter
  - Type Conversion Converter
  - Object Type Converter

Mapping Architecture

To define a mapping, you draw upon the following components:

- The data representation specific to the data source (such as a relational database table or schema-defined XML element) in which you store the object's data.
- A descriptor for a particular object class.
- An object class to map.

A mapping is the same regardless of whether your project is persistent or nonpersistent.

For an example of a typical EclipseLink mapping, see Mapping Examples.

The type of data source you define in your project determines the type of mappings you can use and how you configure them. In a persistent project, you use mappings to persist to a data source. In a nonpersistent project, you use mappings simply to transform between the object format and some other data representation (such as XML).

A descriptor represents a particular domain object: it describes the object's class. A descriptor also owns the mappings: one mapping for each of the class data members that you intend to persist or transform in memory. For more information about descriptors, see Chapter 5, "Understanding Descriptors".

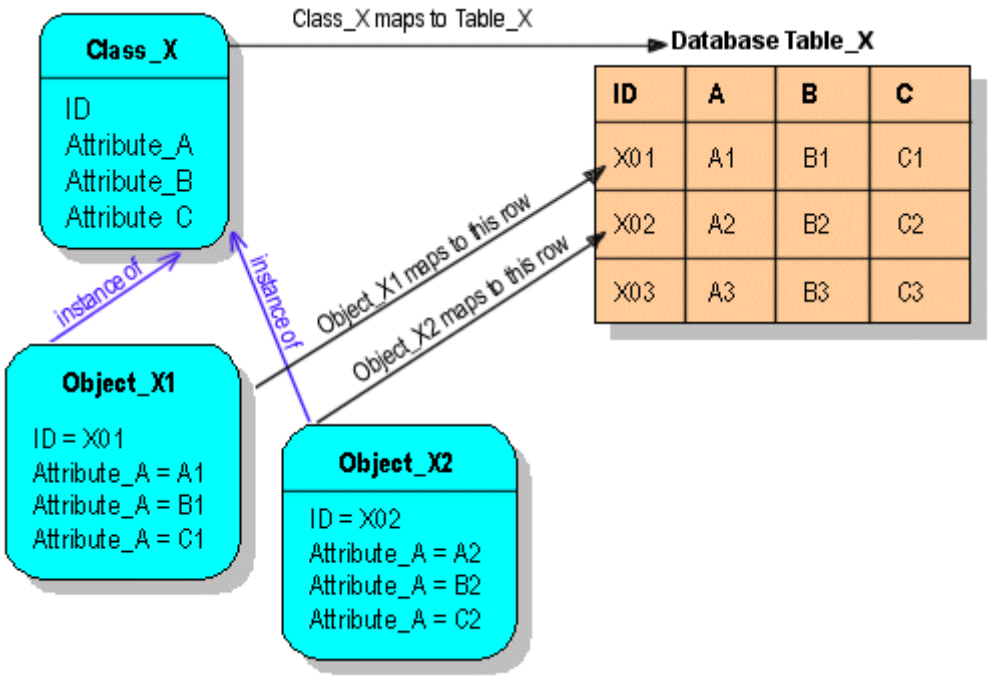
Mapping Examples

Although EclipseLink supports more complex mappings, most EclipseLink classes map to a single database table or XML element that defines the type of information available in the class. Each object instance of a given class maps to a single row comprising the object's attributes, plus an identifier (the primary key) that uniquely identifies the object.

Figure 6-1 illustrates the simplest database mapping case in which:

- **Table_X** in the database represents **Class_X**.
- **Object_X1** and **Object_X2** are instances of **Class_X**.
- Individual rows in **Table_X** represent **Object_X1** and **Object_X2**, as well as any other instances of **Class_X**.

Figure 6-1 How Classes and Objects Map to a Database Table



Description of "Figure 6-1 How Classes and Objects Map to a Database Table"

EclipseLink provides you with the tools to build these mappings, from the simple mappings illustrated in Figure 6-1, to complex mappings.

For an additional example of a relational mapping, see Figure 6-2, "Serialized Object Converter (relational)".

Mapping Converters

If existing EclipseLink mappings do not meet your needs, you can create custom mappings using mapping extensions. These extensions include the following:

- Serialized Object Converter
- Type Conversion Converter
- Object Type Converter

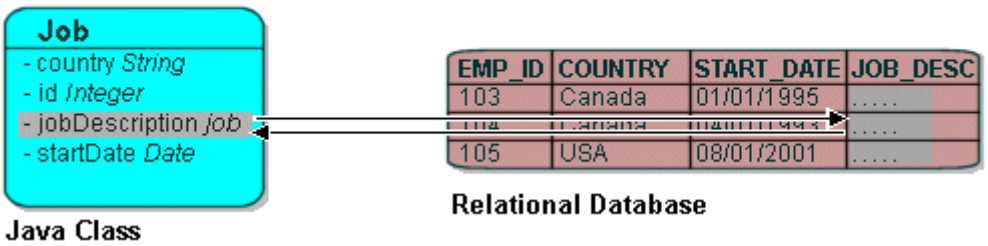
Except for simple type translation, you can use the mapping converters and transformers regardless of whether your data source is relational or nonrelational. Simple type translation is applicable only to XML projects.

Serialized Object Converter

The serialized object converter can be used with direct and direct collection mappings, allowing you to map complex objects into binary fields through Java object serialization. Serialized objects are normally stored in `RAW` or Binary Large Object (`BLOB`) fields in the database, or `HEX` or `BASE64` elements in an XML document.

Figure 6-2 shows an example of a direct-to-field mappings that uses a serialized object converter. The attribute `jobDescription` contains a formatted text document that is stored in the `JOB_DESC` field of the database.

Figure 6-2 Serialized Object Converter (relational)

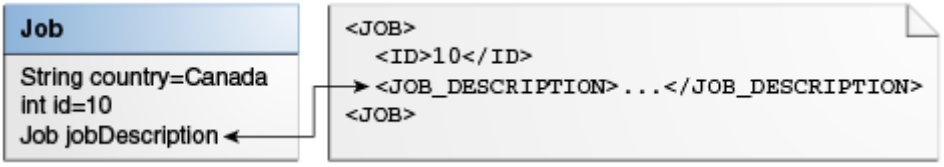


Description of "Figure 6-2 Serialized Object Converter (relational)"

Figure 6-3 demonstrates an example of a nonrelational mapping that uses a serialized object converter. The attribute `jobDescription` contains a formatted text document that EclipseLink stores

in the `JOB_DESCRIPTION` element of an XML schema.

Figure 6-3 Serialized Object Converter (nonrelational)



Description of "Figure 6-3 Serialized Object Converter (nonrelational)"

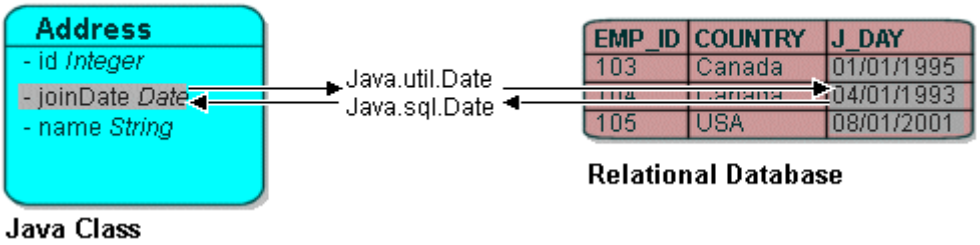
The serialized object converter relies on the Java serializer. Before you map a domain object with the serialized object converter, ensure that the domain object implements the `java.io.Serializable` interface (or inherits that implementation) and marks all nonserializable fields transient.

Type Conversion Converter

The type conversion converter can be used with direct and direct collection mappings, allowing you to map complex objects into binary fields. For example, a `Number` in the data source can be mapped to a `String` in Java, or a `java.util.Date` in Java can be mapped to a `java.sql.Date` in the data source.

Figure 6-4 illustrates a type conversion mapping (relational). Because the `java.util.Date` class is stored by default as a `Timestamp` in the database, it must first be converted to an explicit database type such as `java.sql.Date` (required only for DB2–most other databases have a single date data type that can store any date or time).

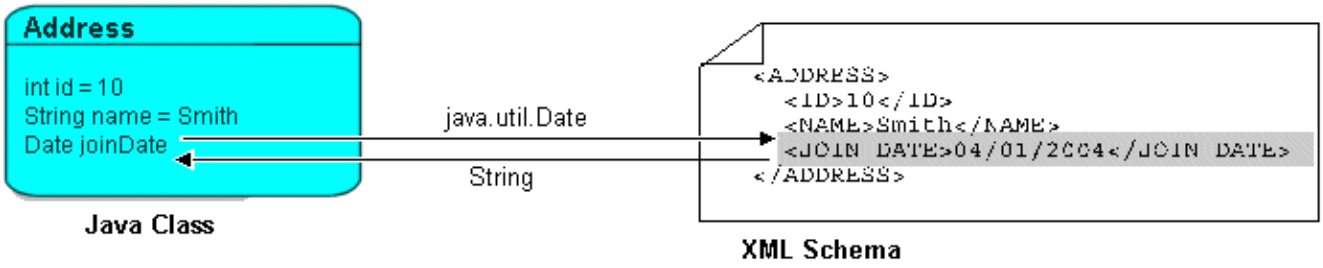
Figure 6-4 Type Conversion Mapping (relational)



Description of "Figure 6-4 Type Conversion Mapping (relational) "

Figure 6-5 illustrates a type conversion mapping (nonrelational). `java.util.Date` object is mapped to a String in a XML schema.

Figure 6-5 Type Conversion Mapping (nonrelational)



Description of "Figure 6-5 Type Conversion Mapping (nonrelational)"

You can use a type conversion converter to specify the specific database type when that type must be handled specially for the database. This includes support for the special Oracle JDBC binding options required for `NCHAR`, `NVARCHAR2`, and `NCLOB` fields as well as the special Oracle Thin JDBC insert and update requirements for handling `BLOB` and `CLOB` fields greater than 5K.

EclipseLink uses the `NCharacter`, `NClob` and `NString` types in the `org.eclipse.persistence.platform.database.oracle` package as the converter data type to support the `NCHAR`, `NCLOB` and `NVARCHAR2` types. EclipseLink uses the `java.sql.Blob` and `Clob` types as the converter data type to support `BLOB` and `CLOB` values greater than 5K.

You can configure a type conversion converter to map a data source time type (such as `TIMESTAMP`) to a `java.lang.String` provided that the String value conforms to the following formats:

- `YYYY/MM/DD HH:MM:SS`
- `YY/MM/DD HH:MM:SS`
- `YYYY-MM-DD HH:MM:SS`
- `YY-MM-DD HH:MM:SS`

For more complex `String` to `TIMESTAMP` type conversion, consider a transformation mapping (see Transformation Mapping).

You can also use the `@TypeConverter` annotation to modify data values during the reading and writing of a mapped attribute. Each `TypeConverter` must be uniquely named and can be defined at the class, field, and property level, and can be specified within an `Entity`, `MappedSuperclass` and `Embeddable` class. A `TypeConverter` is always specified by using an `@Convert` annotation.

You can place a `@TypeConverter` on a `Basic`, `BasicMap`, or `BasicCollection` mapping. For more information on these annotations, see *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

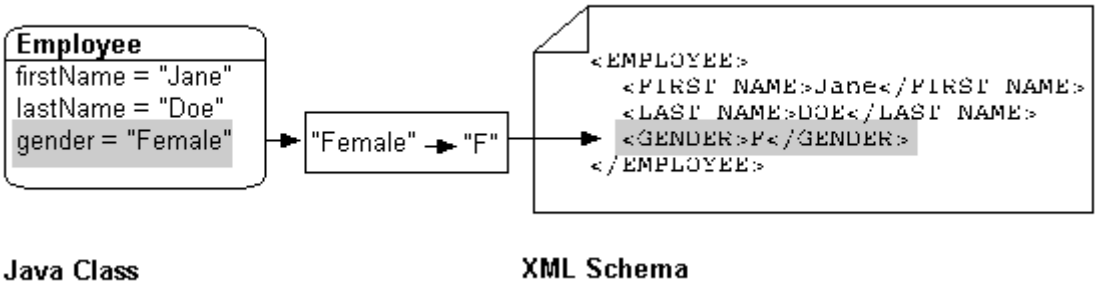
Object Type Converter

The object type converter can be used with direct and direct collection mappings allowing you to match a fixed number of values to Java objects. Use this converter when the values in the schema differ from those in Java.

Figure 6-6 illustrates an object type conversion between the `Employee` attribute `gender` and the XML element `gender`. If the value of the Java object attribute is `Female`, EclipseLink stores it in the XML

element as F.

Figure 6-6 Object Type XML Converter



Description of "Figure 6-6 Object Type XML Converter"

You can also perform object type transformations by using the `@ObjectTypeConverter` annotation. This annotation specifies an `org.eclipse.persistence.mappings.converters.ObjectTypeConverter` that converts a fixed number of database data value(s) to Java object value(s) during the reading and writing of a mapped attribute. For this annotation you must provide values for the array of conversion values by using the `@ConversionValue` annotation. For more information, see the descriptions of `@ObjectTypeConverter` and `@ConversionValue` in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

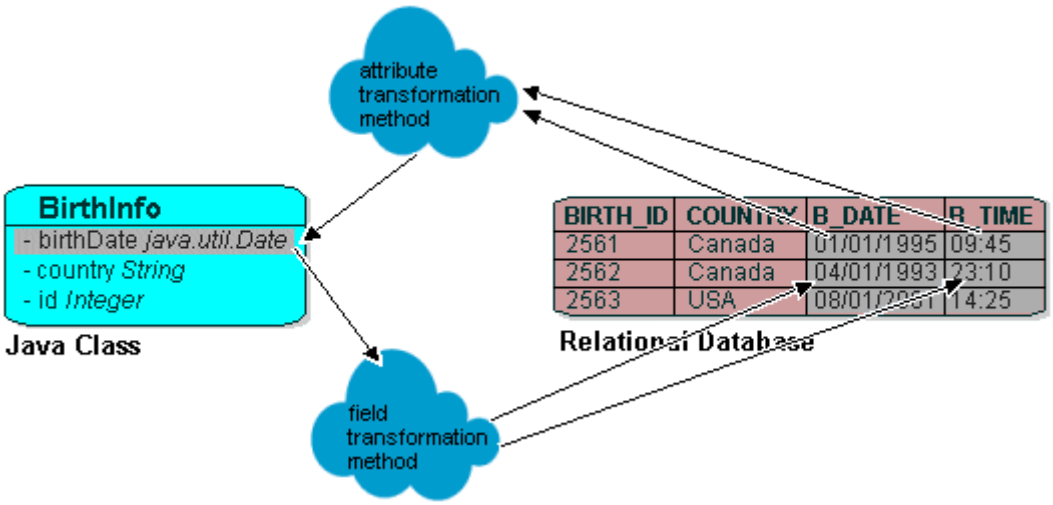
Transformation Mapping

In some special circumstances, existing mapping types and their default Java to data source type handling may be insufficient. In these special cases, you can consider using a transformation mapping to perform specialized translations between how a value is represented in Java and in the data source.

Because of the complexity of transformation mappings, it is often easier to perform the transformation with a converter or getter and setter methods of a direct-to-field mapping.

Figure 6-7 illustrates a transformation mapping. The values from the `B_DATE` and `B_TIME` fields are used to create a `java.util.Date` to be stored in the `birthDate` attribute.

Figure 6-7 Transformation Mappings



Description of "Figure 6-7 Transformation Mappings"

A transformation mapping is made up of the following two components:

- attribute transformer: performs the object attribute transformation at read time
- field transformer: performs the object attribute-to-field transformation at write time

You can implement a transformer as either a separate class or as a method on your domain object.

Often, a transformation mapping is appropriate when values from multiple fields are used to create an object. This type of mapping requires that you provide an *attribute transformation* that is invoked when reading the object from the database. This must have at least one parameter that is an instance of `Record`. In your attribute transformation, you can use `Record` method `get` to retrieve the value in a specific column. Your attribute transformation can optionally specify a second parameter, an instance of `Session`. The `Session` performs queries on the database to get additional values needed in the transformation. The transformation should *return* the value to be stored in the attribute.

Transformation mappings also require a *field transformation* for each field, to be written to the database when the object is saved. The transformation returns the value to be stored in that field.

Within your implementation of the attribute and field transformation, you can take whatever actions are necessary to transform your application data to suit your data source, and vise versa.

You can perform transformation mappings between database columns and attribute values by using the `@Transformation` annotation. Use this annotation with the `@WriteTransformer` and `@ReadTransformer` annotations. The `@WriteTransformer` annotation is used to transform a single attribute value to a single database column value. For this annotation you have the option of providing an implementation of the `FieldTransformer` interface. For the `@ReadTransformer` annotation, you must provide an implementation of the `org.eclipse.persistence.mappings.transformers.AttributeTransformer` interface. For more information on these annotations, see the descriptions of the `@Transformation`, `@ReadTransformer`, and `@WriteTransformer` in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

6.2. Object-Relational Mapping Concepts

This section describes concepts for relational mappings that are unique to EclipseLink:

- Indirection (Lazy Loading)
- Indirection, Serialization, and Detachment
- Value Holder Indirection
- Transparent Indirection
- Proxy Indirection
- Weaved Indirection
- About JPA Mapping Types

Indirection (Lazy Loading)

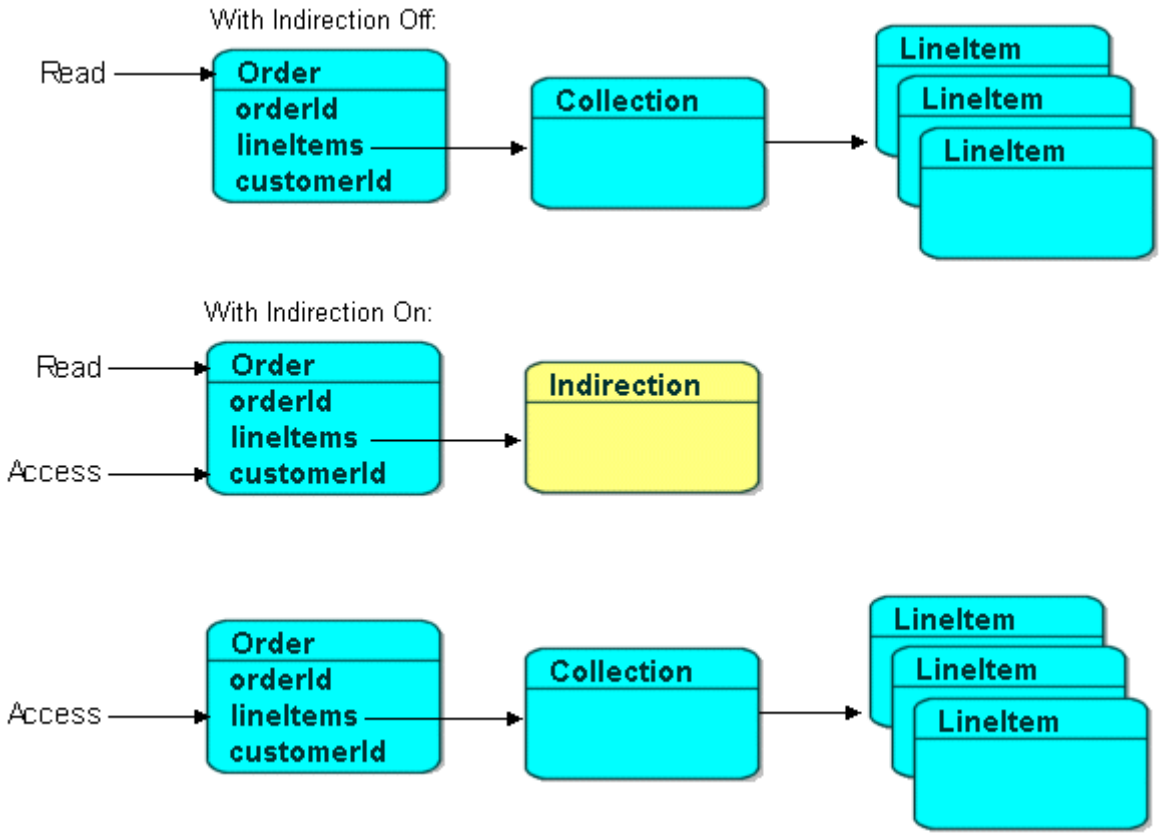
By default, when EclipseLink retrieves a persistent object, it retrieves all of the dependent objects to which it refers. When you configure indirection (also known as lazy reading, lazy loading, and just-in-time reading) for an attribute mapped with a relationship mapping, EclipseLink uses an indirection object as a place holder for the referenced object: EclipseLink defers reading the dependent object until you access that specific attribute. This can result in a significant performance improvement, especially if the application is interested only in the contents of the retrieved object, rather than the objects to which it is related.

Oracle strongly recommends using indirection for all relationship mappings. Not only does this lets you optimize data source access, but it also allows EclipseLink to optimize the persistence unit processing, cache access, and concurrency.

- The use of indirection is especially important for providing a proper maintenance of bidirectional relationships. In this case, you must use indirection. If you are operating with collections, you must use transparent indirection (see Transparent Indirection).
- The implementation of indirection (lazy loading) is vendor-specific. Serializing entities and merging those entities back into a persistence context may not be interoperable across vendors when lazy properties or fields and/or relationships are used.

Figure 6-8 shows an indirection example. Without indirection, reading the `Order` object also reads the dependent collection of `LineItem` objects. With indirection, reading the `Order` object does not read the dependent collection of `LineItem` objects: the `lineItems` attribute refers to an indirection object. You can access other attributes (such as `customerId`), but EclipseLink reads the dependent `LineItem` objects only if and when you access the `lineItems` attribute.

Figure 6-8 EclipseLink Indirection



Description of "Figure 6-8 EclipseLink Indirection"

EclipseLink supports the following types of indirection:

- Value Holder Indirection
- Transparent Indirection
- Proxy Indirection

When using indirection with an object that your application serializes, you must consider the effect of any untriggered indirection objects at deserialization time. See Indirection, Serialization, and Detachment.

Indirection, Serialization, and Detachment

When using indirection (lazy loading), it is likely that a graph of persistent objects will contain untriggered indirection objects. Because indirection objects are transient and do not survive serialization between one JVM and another, untriggered indirection objects will trigger an error if the relationship is accessed after deserialization.

The application must ensure that any indirect relationships that will be required after deserialization have been instantiated before serialization. This can be done through accessing the get method for any relationship using `ValueHolder` or weaved indirection, and by calling the `size` method to any relationship using transparent indirection. If the application desired the relationships to be always instantiated on serialization, you could overwrite the serialization `writeObject` method in the persistent class to first instantiate the desired relationships. Use caution for objects with many or deep relationships to avoid serializing large object graphs: ideally, only the

relationships required by the client should be instantiated.

When serializing JPA entities, any lazy relationships that have not been instantiated prior to serialization will trigger errors if they are accessed. If weaving is used on the server, and the entities are serialized to a client, the same weaved classes must exist on the client, either through static weaving of the jar, or through launching the client JVM using the EclipseLink agent.

For more information, see Using Java Byte-code Weaving.

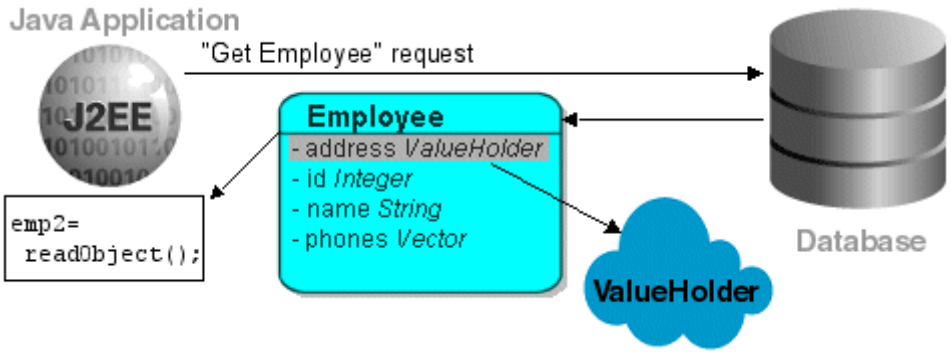
Value Holder Indirection

Persistent classes that use indirection must replace relationship attributes with value holder attributes. A value holder is an instance of a class that implements the `ValueHolderInterface` interface, such as `ValueHolder`. This object stores the information necessary to retrieve the object it is replacing from the database. If the application does not access the value holder, the replaced object is never read from the database.

To obtain the object that the value holder replaces, use the `getValue` and `setValue` methods of the `ValueHolderInterface`. A convenient way of using these methods is to hide the `getValue` and `setValue` methods of the `ValueHolderInterface` inside `get` and `set` methods, as shown in the following illustrations.

Figure 6-9 shows the `Employee` object being read from the database. The `Address` object is not read and will not be created unless it is accessed.

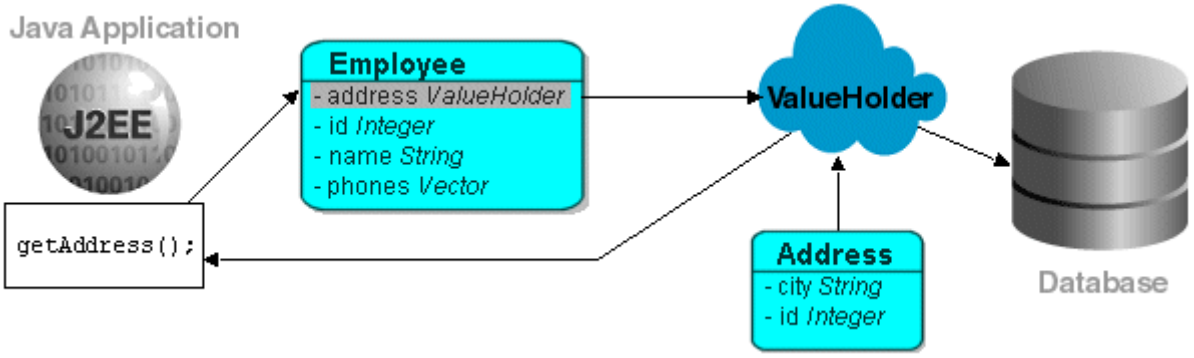
Figure 6-9 Address Object Not Read



Description of "Figure 6-9 Address Object Not Read"

The first time the address is accessed, as in Figure 6-10, the `ValueHolder` reads and returns the `Address` object.

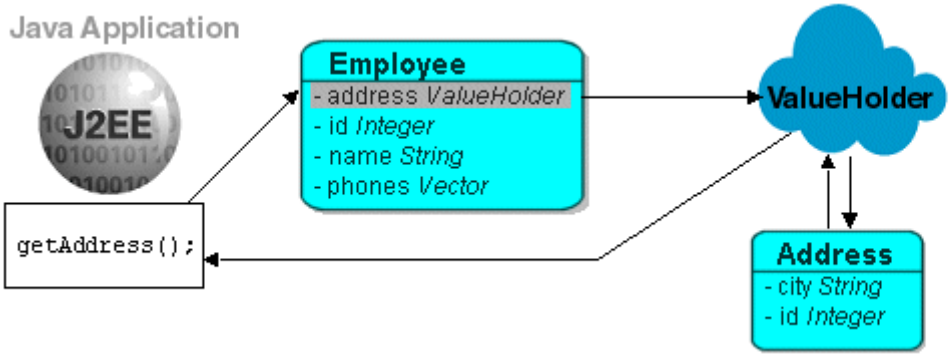
Figure 6-10 Initial Request



Description of "Figure 6-10 Initial Request"

Subsequent requests for the address do not access the database, as shown in Figure 6-11.

Figure 6-11 Subsequent Requests



Description of "Figure 6-11 Subsequent Requests"

If you are using method access, the get and set methods specified in the mapping must access the instance of `ValueHolderInterface`, rather than the object referenced by the value holder. The application should not use these getter and setter, but use the getter and setter that hide the usage of value holders.

Transparent Indirection

Transparent indirection lets you declare any relationship attribute of a persistent class that holds a collection of related objects as any of the following Java objects:

- `java.util.Collection`
- `java.util.Hastable`
- `java.util.List`
- `java.util.Map`
- `java.util.Set`
- `java.util.Vector`

EclipseLink will use an indirection object that implements the appropriate interface and also performs just-in-time reading of the related objects. When using transparent indirection, you do

not have to declare the attributes as `ValueHolderInterface`.

Newly created collection mappings use transparent indirection by default if their attribute *is not* a `ValueHolderInterface`.

You can configure EclipseLink to automatically weave transparent indirect container indirection for JPA entities and Plain Old Java Object (POJO) classes. For more information, see Using Java Byte-code Weaving and About Weaving.

Proxy Indirection

The Java class `Proxy` lets you use dynamic proxy objects as place-holders for a defined interface. Certain EclipseLink mappings can be configured to use proxy indirection, which gives you the benefits of indirection without the need to include EclipseLink classes in your domain model. Proxy indirection is to one-to-one relationship mappings as indirect containers are to collection mappings.

To use proxy indirection, your domain model must satisfy all of the following criteria:

- The target class of the one-to-one relationship must implement a public interface.
- The one-to-one attribute on the source class must be of the `interface` type.
- If you employ method accessing, then the getter and setter methods must use the interface.

Before using proxy indirection, be aware of the restrictions it places on how you use the persistence unit (see Proxy Indirection Restrictions).

To configure proxy indirection, you can use JDeveloper or Java in an amendment method.

Proxy Indirection Restrictions

Proxy objects in Java are only able to intercept messages sent. If a primitive operation such as `==`, `instanceof`, or `getClass` is used on a proxy, it will not be intercepted. This limitation can require the application to be somewhat aware of the usage of proxy objects.

You cannot register the target of a proxy indirection implementation with a persistence unit. Instead, first register the source object with the persistence unit. This lets you retrieve a target object clone with a call to a getter on the source object clone.

Weaved Indirection

For JPA entities or POJO classes that you configure for weaving, EclipseLink weaves value holder indirection for one-to-one mappings. If you want EclipseLink to weave change tracking and your application includes collection mappings (one-to-many or many-to-many), then you must configure all collection mappings to use transparent indirect container indirection only (you may not configure your collection mappings to use eager loading nor value holder indirection).

For more information, see Using Java Byte-code Weaving.

About JPA Mapping Types

To map entity classes to relational tables you must configure a mapping per persistent field. The following sections describe EclipeLink's JPA mapping types:

- Basic Mappings
- Default Conversions and Converters
- Collection Mappings
- Using Optimistic Locking

Basic Mappings

Simple Java types are mapped as part of the immediate state of an entity in its fields or properties. Mappings of simple Java types are called basic mappings.

By default, the EclipseLink persistence provider automatically configures a basic mapping for simple types.

Use the following annotations to fine-tune how your application implements these mappings:

- `@Basic`
- `@Enumerated`
- `@Temporal`
- `@Lob`
- `@Transient`
- `@Column`
- Lazy Basics (See Using Indirection with Collections)

For all mapping types there are a common set of options:

- Read-Only: Specifies that the mapping should populate the value on read and copy. Required when multiple mappings share the same database column.
- Converters: Allows custom data types and data conversions to be used with most mapping types
  - Annotations: `@Converter`, `@TypeConverter`, `@ObjectTypeConverter`, `@StructConverter`, `@Convert`
  - External Metadata: `<converter>`, `<type-converter>`, `<object-type-converter>`, `<struct-converter>`, `<convert>`

For more information on these annotations, see *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Default Conversions and Converters

The section "Converter Annotations" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink* provides a list of the converter annotation extensions defined by EclipseLink and links to their descriptions.

See the individual converter annotations in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink* for descriptions of the following:

- the order in which the EclipseLink persistence provider searches the converter annotations
- the types of classes for which you can specify converters (you can define converters at the class, field and property level)
- the mappings with which you can use converters

Collection Mappings

You can access additional advanced mappings and mapping options through the EclipseLink descriptor and mapping API using a `DescriptorCustomizer` class.

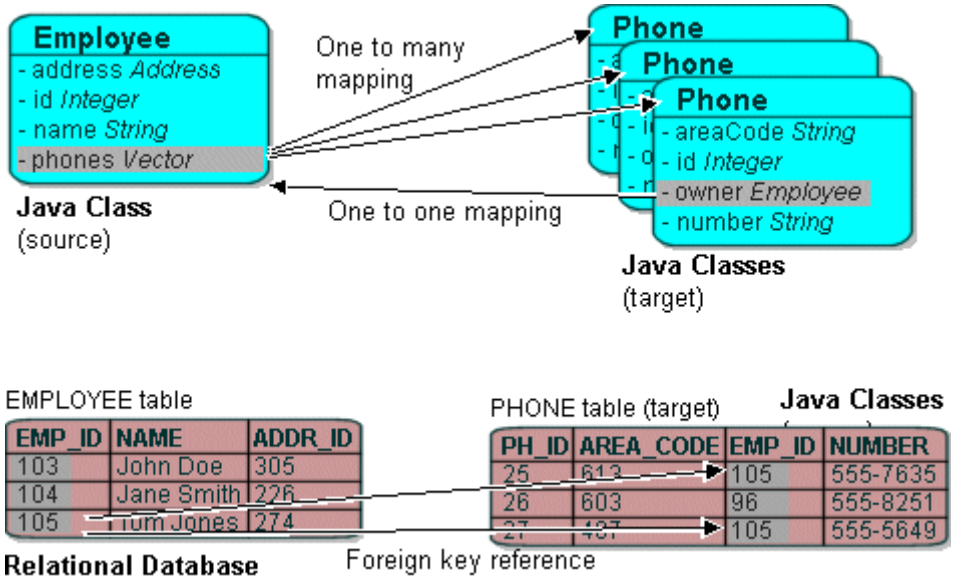
- One-to-Many Mapping
- Many-to-Many Mapping
- Using Indirection with Collections

One-to-Many Mapping

One-to-many mappings are used to represent the relationship between a single source object and a collection of target objects. They are a good example of something that is simple to implement in Java using a Collection (or other collection types) of target objects, but difficult to implement using relational databases.

In a Java Collection, the owner references its parts. In a relational database, the parts reference their owner. Relational databases use this implementation to make querying more efficient.

Figure 6-12 One-to-Many Relationships



Description of "Figure 6-12 One-to-Many Relationships"

The phone attribute shown in the One-to-Many Relationships is of type Vector. You can use a Collection interface (or any class that implements the Collection interface) for declaring the collection attribute.

JPA Mapping

By default, JPA automatically defines a OneToMany mapping for a many-valued association with one-to-many multiplicity.

Use the `@OneToMany` annotation to do the following:

- configure the fetch type to `EAGER`
- configure the associated target entity, because the Collection used is not defined using generics
- configure the operations that must be cascaded to the target of the association: for example, if the owning entity is removed, ensure that the target of the association is also removed
- configure the details of the join table used by the persistence provider for unidirectional one-to-many relationships. For a one-to-many using a `mappedBy` or `JoinColumn`, the deletion of the related objects is cascaded on the database. For a one-to-many using a `JoinTable`, the deletion of the join table is cascaded on the database (target objects cannot be cascaded even if private because of constraint direction).

For more information, see Section 11.1.23 "JoinTable Annotation" in the JPA Specification.

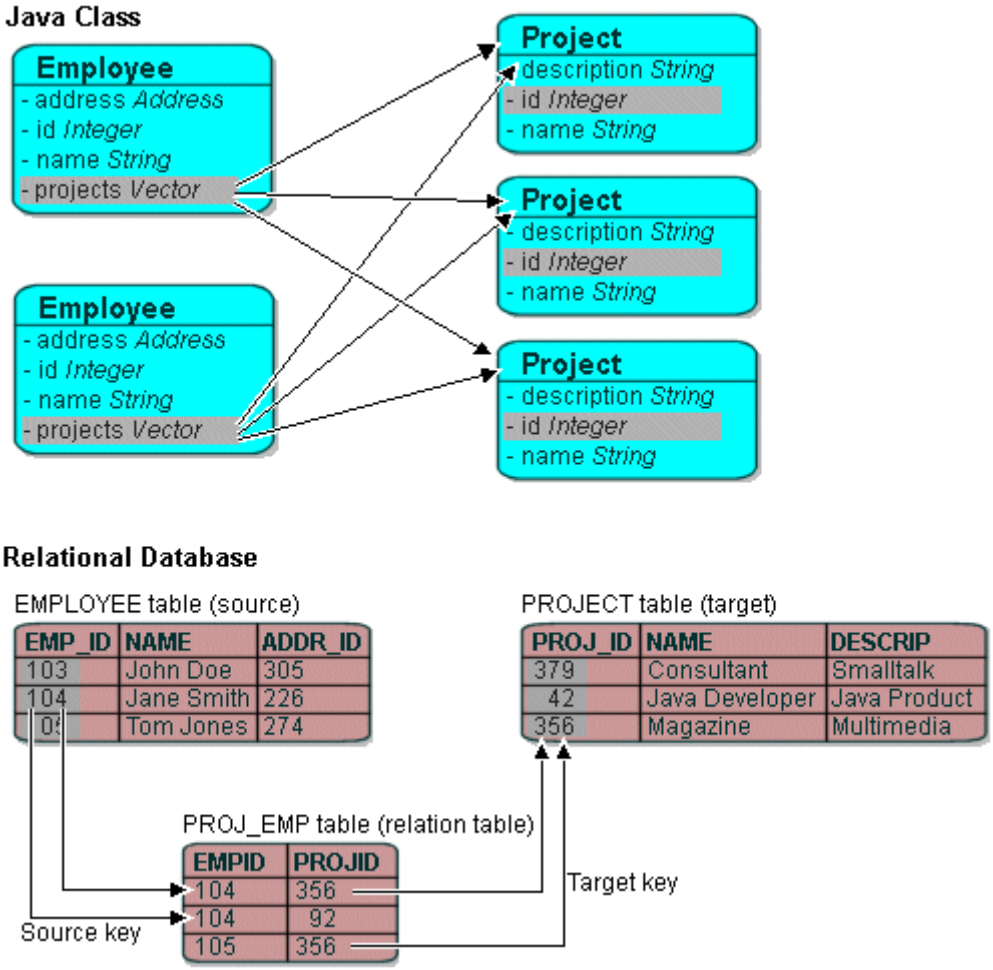
`http://jcp.org/en/jsr/detail?id=338`

Many-to-Many Mapping

Many-to-many mappings represent the relationships between a collection of source objects and a collection of target objects. They require the creation of an intermediate table for managing the associations between the source and target records.

Figure 6-13 illustrates a many-to-many mapping in Java and in relational database tables.

Figure 6-13 Many-to-Many Relationships



Description of "Figure 6-13 Many-to-Many Relationships"

For the projects attribute shown in the Many-to-many Relationships you can use a `Collection` interface (or any class that implements the `Collection` interface) for declaring the collection attribute.

JPA Mapping

By default, JPA automatically defines a many-to-many mapping for a many-valued association with many-to-many multiplicity.

Use the `@ManyToMany` annotation to do the following:

- configure the `FetchType` to `EAGER`
- configure the mapping to forbid null values (for nonprimitive types) in case null values are inappropriate for your application
- configure the associated target entity because the Collection used is not defined using generics

- configure the operations that must be cascaded to the target of the association (for example, if the owning entity is removed, ensure that the target of the association is also removed)

For a list of supported attributes for the `@ManyToMany` annotation, see the Java Persistence specification:

`http://jcp.org/en/jsr/detail?id=338`

Using Indirection with Collections

JPA specifies that lazy loading is a hint to the persistence provider that data should be fetched lazily when it is first accessed, if possible. If you are developing your application in a Jakarta EE environment, set fetch to `jakarta.persistence.FetchType.LAZY`, and the persistence provider supplies all the necessary functionality.

When using a one-to-one or many-to-one mapping in a Java SE environment, use either dynamic or static weaving to perform lazy loading when the `fetch` attribute is set to `FetchType.LAZY`. Also in the Java SE environment, one-to-many and many-to-many relationships are lazy by default and use transparent indirection, while one-to-one and many-to-one relationships are not lazy.

When using a one-to-one or many-to-one mapping in a Java SE environment and the environment does not permit the use of `-javaagent` on the JVM command line, use static weaving to perform lazy loading when the `fetch` attribute is set to `FetchType.LAZY`.

If you set one-to-one or many-to-one relationships to lazy, and you enable weaving, the EclipseLink JPA persistence provider will use weaving to enable value holder indirection for these relationships.

The collection annotations `@OneToOne`, `@OneToMany`, `@ManyToMany`, and `@ManyToOne` provide a `fetch` mapping attribute which can be set to `lazy` or `eager`. When you set the attribute to `lazy`, the EclipseLink JPA persistence provider uses indirection.

Table 6-1 lists support for lazy loading by mapping type.

Table 6-1 Support for Lazy Loading by Mapping Type

Mapping	Jakarta EE	Java SE
Many-to-many	Lazy loading is performed when the `fetch` attribute is set to `jakarta.persistence.FetchType.LAZY` (default).	Lazy loading is performed when the `fetch` attribute is set to `jakarta.persistence.FetchType.LAZY` (default).

One-to-many	Lazy loading is performed when the `fetch` attribute is set to `jakarta.persistence.FetchType.LAZY` (default).	Lazy loading is performed when the `fetch` attribute is set to `jakarta.persistence.FetchType.LAZY` (default).
One-to-one	Lazy loading is performed when the `fetch` attribute is set to `jakarta.persistence.FetchType.LAZY`.	The `fetch` attribute is ignored and default `jakarta.persistence.FetchType.EAGER` applies.
Many-to-one	Lazy loading is performed when the `fetch` attribute is set to `jakarta.persistence.FetchType.LAZY`.	The `fetch` attribute is ignored and default `jakarta.persistence.FetchType.EAGER` applies
Basic	Lazy loading is performed when the `fetch` attribute is set to `jakarta.persistence.FetchType.LAZY`.	The `fetch` attribute is ignored and default `jakarta.persistence.FetchType.EAGER` applies

Using Optimistic Locking

Oracle recommends using optimistic locking. With optimistic locking, all users have read access to the data. When a user attempts to write a change, the application checks to ensure the data has not changed since the user read the data.

Optimistic Locking in a Stateless Environment

In a stateless environment, take care to avoid processing out-of-date (stale) data. A common strategy for avoiding stale data is to implement optimistic locking, and store the optimistic lock values in the object. This solution requires careful implementation if the stateless application serializes the objects, or sends the contents of the object to the client in an alternative format. In this case, transport the optimistic lock values to the client in the HTTP contents of an edit page. You must then use the returned values in any write transaction to ensure that the data did not change while the client was performing its work.

You can use optimistic version locking or optimistic field locking policies. Oracle recommends using version locking policies.

Optimistic Version Locking

Use the `@Version` annotation to enable the JPA-managed optimistic locking by specifying the version field or property of an entity class that serves as its optimistic lock value (recommended).

When choosing a version field or property, ensure that the following is true:

- there is only one version field or property per entity
- you choose a property or field persisted to the primary table
- your application does not modify the version property or field

For more information, see Section 11.1.45 "Table Annotation" in the JPA Specification.

http://jcp.org/en/jsr/detail?id=338

```
 NOTE: The field or property type must either be a numeric type (such as
`Number`, `long`, `int`, `BigDecimal`, and so on), or a
`java.sql.Timestamp`. EclipseLink recommends using a numeric type.
```

The `@Version` annotation does not have attributes. The `@Version` annotation allows you to use EclipseLink converters. See Default Conversions and Converters.

For more information, see Section 11.1.9 "Column Annotation" in the JPA Specification.

http://jcp.org/en/jsr/detail?id=338

6.3. MOXy Mapping Concepts

XML mappings transform object data members to the XML elements of an XML document whose structure is defined by an XML Schema Document (XSD). You can map the attributes of a Java object to a combination of XML simple and complex types using a wide variety of XML mapping types.

Classes are mapped to complex types, object relationships map to XML elements, and simple attributes map to text nodes and XML attributes. The real power in using MOXy is that when mapping an object attribute to an XML document, XPath statements are used to specify the location of the XML data.

EclipseLink stores XML mappings for each class in the class descriptor. EclipseLink uses the descriptor to instantiate objects mapped from an XML document and to store new or modified objects as XML documents.

EclipseLink provides XML mappings that are not defined in the JAXB specification. Some of the MOXy extensions are available through EclipseLink annotations; others require programmatic changes to the underlying metadata.

Mapping concepts for MOXy are described in *Developing JAXB Applications EclipseLink MOXy*. See the following chapters:

"EclipseLink MOXy Runtime" describes:

- the EclipseLink XML Bindings document, which is an alternative to the JAXB annotations. Not only can XML Bindings separate your mapping information from your actual Java class, it can also be used for more advanced metadata
- the several different bootstrapping options that you can use when creating your `JAXBContext`.
- the `MetadataSource` interface, which is responsible for serving up EclipseLink metadata.

Providing an implementation of this interface allows you to store mapping information outside of your application and have it retrieved when the application's `JAXBContext` is being created or refreshed.

- schema generation and validation.
- the several mechanisms by which you can get event callbacks during the marshalling and unmarshalling processes. You can specify callback methods directly on your mapped objects, or define separate `Listener` classes and register them with the JAXB runtime
- querying objects by XPath. This is an alternative to using conventional Java access methods to get and set your object's values. EclipseLink MOXy allows you to access values using an XPath statement. There are special APIs on EclipseLink's `JAXBContext` to allow you to get and set values by XPath.
- the use of the JAXB `Binder` interface, which allows you to preserve an entire XML document, even if only some of the items are mapped.

"Mapping Type Levels" describes the initial tasks of setting up a mapping in MOXy:

- the default root element, which tells EclipseLink what the top-level root of your XML document will be.
- namespace information for the Java class, and that all of the elements must be qualified for the namespace. You can namespace-qualify the elements on the package, type, or field/property level.
- the ways in which you can specify inheritance hierarchy in XML by using `xsi:type` attribute, substitution groups, or the MOXY-specific `@XmlDiscriminatorNode` and `@XmlDiscriminatorValue` annotations

"Mapping Simple Values" and "Mapping Special Schema Types" describes how Java values can be mapped to XML in several different ways:

- Java values can be mapped to XML attributes, text nodes, schema types, or simple type translators
- collections of simple Java values can be mapped to text nodes, text nodes within a grouping element, list elements, or a collection of `XmlAttributes` or `XmlValues`
- multiple Java mappings can be created for a single property using OXM metadata, with the caveat that at most one mapping will be readable (the rest will be "write-only")
- Java `enums` can be mapped to XML using the `@XmlEnum` and `@XmlEnumValue` annotations
- dates and time: EclipseLink MOXy supports the following types which are not covered in the JAXB specification: `java.sql.Date` `java.sql.Time` , and `java.sql.Timestamp`.
- union files: When EclipseLink unmarshalls the XML document, such as an XML Schema Union, it tries each of the union types until it can make a successful conversion. Currently, EclipseLink does not support the mapping of Unions using Annotations or OXM Metadata. However, an EclipseLink XML Customizer can be used to create the mapping.
- binary types: EclipseLink supports marshalling and unmarshalling binary data in two different representation formats: `base64Binary` (default) and `hexBinary`. You can specify the desired binary format using the `@XmlSchemaType` annotation, or `<xml-schema-type>` element in EclipseLink OXM.

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.

Mapping Values

There are several ways to map simple Java values and collections of simple values directly to XML text nodes. You can map to attributes, text nodes, or schema types. You can also use simple type translators to map types of nodes that are not defined in your XML schema. These techniques are described in "Mapping Simple Values" in *Developing JAXB Applications EclipseLink MOXy*.

6.4. Object-JSON Mapping Concepts

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.

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

EclipseLink provides the following support for mapping JSON documents:

- JSON bindings that do not require compile time dependencies, in addition to those required for normal JAXB use. You can also write MOXy External Bindings files as JSON documents.
- Although XML has a single datatype, JSON differentiates between strings, numbers, and booleans. EclipseLink supports these datatypes automatically.
- 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.
- 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. With EclipseLink, you can override this behavior (that is, omit the root element from the JSON output).
- Because JSON does not use namespaces, all namespaces and prefixes are ignored by default when marshaling and unmarshaling. 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.
- By default, when marshalling to JSON, EclipseLink marshals empty collections as [ ], EclipseLink allows you to override this behavior, so that empty collections are not marshalled at all.
- You can marshal and unmarshal root-level collections.

For more information on EclipseLink support for JSON documents, see "Using JSON Documents" in *Developing JAXB Applications EclipseLink MOXy*

Chapter 7. Understanding Data Access

This chapter describes one of the most important functions of a session, which is to provide access to a data source. This chapter describes the concepts behind data access within a session that are unique to EclipseLink.

This chapter includes the following sections:

7.1. About Externally Managed Transactional Data Sources

EclipseLink transactional data sources are *externally managed* if the connection pool is managed by a transaction service (such as an application server controlled transaction or a JTA transaction). A JTA managed data source or connection pool is commonly used in Jakarta EE applications and normally required in EJB applications. Use an externally-managed connection pool as follows:

- Configure the session to use an `ExternalTransactionController` to integrate EclipseLink's persistence unit with the external transaction service. Use the `eclipselink.target-server persistence` unit property to configure the name of the class that implements the `ExternalTransactionController` interface.
- Use the boolean `external-transaction-controller` option on the persistence unit to indicate that the transaction is managed by a transaction manager and should not be managed by EclipseLink. This can also be used if the datasource does not support transactions to specify the connection's login and inform EclipseLink that the connection is maintained by the external controller.
- You may need to configure the EclipseLink read connection pool or sequence connection pool to use a non-JTA connection pool in order to avoid transactional overhead. For more information, see "Default (Write) and Read Connection Pools" and "Sequence Connection Pools".

7.2. About Data Source Login Types

The login (if any) associated with a session determines how the EclipseLink runtime connects to the project's data source. For projects that do not persist to a data source, a login is not required. For

projects that do persist to a data source, a login is always required. A login includes details of data source access, such as authentication, use of connection pools, and use of external transaction controllers. A `Login` owns a data source platform.

A data source platform includes options specific to a particular data source including binding, use of native SQL, use of batch writing, and sequencing.

You can use a login in a variety of roles. A login's role determines where and how you create it. The login role you choose depends on the type of project you are creating and how you intend to use the login.

There is a session login type for each project type that persists to a data source.

Note that there is no XML login. EclipseLink XML projects are used for nonpersistent, in-memory object to XML data transformation and consequently there is no data source to log in to.

If you are creating a project that accesses a relational database, you must configure the project with a `DatabaseLogin`. Your choice of `DatabasePlatform` further customizes your project for a particular type of database. can be configured through the `datasource-login` persistence unit option. This option contains attributes for configuring user name, password, the name of the class implementing the data source platform, and others.

7.3. About Data Source Platform Types

EclipseLink abstracts the details of your underlying data source using data source platform classes. A data source platform is owned by your project's `Login`. Specify your database platform at the project level for all sessions, or override this project-level configuration at the session level.

To configure most platform options, you must use an amendment method, or a `preLogin` event listener.

EclipseLink interacts with databases using structured query language (SQL). Because each database platform uses its own variation on the basic SQL language, EclipseLink must adjust the SQL it uses to communicate with the database to ensure that the application runs smoothly.

The type of database platform you choose determines the specific means by which the EclipseLink runtime accesses the database, including the type of Java Database Connectivity (JDBC) driver to use. JDBC is an application programming interface (API) that gives Java applications access to a database. EclipseLink relational projects rely on JDBC connections to read objects from, and write objects to, the database. EclipseLink applications use either individual JDBC connections or a JDBC connection pool, depending on the application architecture.

The `DatabasePlatform` class encapsulates behavior specific to a database platform (such as, Oracle, Sybase, DBase), and provides the protocol for EclipseLink to access this behavior

EclipseLink provides a variety of database-specific platforms that let you customize your project for your target database. For a list of supported database platforms, see `org.eclipse.persistence.config.TargetDatabase` class and Database Support.

A list of supported application servers can be found in

`org.eclipse.persistence.config.TargetServer` and Application Server Support. The name of the application server you wish to use can be specified with the `eclipselink.target-server.persistence` unit property.

The `datasource-login` persistence unit option contains other data source properties you can configure, such as user name, password, connection pooling, and so on. You can implement the `Login` class, and then set properties specific to database.

7.4. About Authentication

Authentication is the means by which a data source validates a user's identity and determines whether the user has sufficient privileges to perform a given action. Authentication plays a central role in data security and user accountability and auditing.

For two-tier applications, simple JDBC authentication is usually sufficient.

The following sections describe the different authentication strategies:

- Simple JDBC Authentication
- Oracle Database Proxy Authentication
- Auditing

Simple JDBC Authentication

When you configure an EclipseLink database login with a user name and password, EclipseLink provides these credentials to the JDBC driver that you configure your application to use.

By default, EclipseLink reads passwords from the `persistence.xml` file.

Oracle Database Proxy Authentication

EclipseLink supports proxy authentication with Oracle Database in Java SE applications and Jakarta EE applications with the Oracle JDBC driver and external connection pools only.

EclipseLink does not support Oracle Database proxy authentication with JTA.

Oracle Database proxy authentication delivers the following security benefits:

- A limited trust model, by controlling the users on whose behalf middle tiers can connect, and the roles the middle tiers can assume for the user.
- Scalability, by supporting user sessions through Oracle Call Interface (OCI) and thick JDBC, and eliminating the overhead of reauthenticating clients.
- Accountability, by preserving the identity of the real user through to the database, and enabling auditing of actions taken on behalf of the real user.

- Flexibility, by supporting environments in which users are known to the database, and in which users are merely "application users" of which the database has no awareness.

Oracle Database supports proxy authentication in three-tiers only; it does not support it across multiple middle tiers.

For more information about authentication in Oracle Database, see "Preserving User Identity in Multitiered Environments" in the *Oracle Database Security Guide*.

Configure your EclipseLink database login to use proxy authentication to do the following:

- address the complexities of authentication in a three-tier architecture (such as client-to-middle-tier and middle-tier-to-database authentication, and client reauthentication through the middle -tier to the database)
- enhance database audit information (for even triggers and stored procedures) by using a specific user for database operations, rather than the generic pool user
- simplify VPD/OLS configuration by using a proxy user, rather than setting user information directly in the session context with stored procedures

Auditing

Regardless of what type of authentication you choose, EclipseLink logs the name of the user associated with all database operations. Example 7-1 shows the `CONFIG` level EclipseLink logs when a `ServerSession` connects through the main connection for the sample user "scott", and a `ClientSession` uses proxy connection "jeff"

Example 7-1 Logs with Oracle Database Proxy Authentication

```
[EclipseLink Config]--ServerSession(13)--Connection(14)--Thread(Thread[main,5,main])--
connecting(DatabaseLogin( platform=>Oracle9Platform   user name=> "scott"
connector=>OracleJDBC10_1_0_2ProxyConnector datasource name=>DS))
[EclipseLink Config]--ServerSession(13)--Connection(34)--Thread(Thread[main,5,main])--
Connected: jdbc:oracle:thin:@localhost:1521:orcl
User: SCOTT
[EclipseLink Config]--ClientSession(53)--Connection(54)--Thread(Thread[main,5,main])--
connecting(DatabaseLogin(platform=>Oracle9Platform user name=> "scott"
connector=>OracleJDBC10_1_0_2ProxyConnector datasource name=>DS))
[EclipseLink Config]--ClientSession(53)--Connection(56)--Thread(Thread[main,5,main])--
Connected: jdbc:oracle:thin:@localhost:1521:orcl
User: jeff
```

Your database server likely provides additional user auditing options. Consult your database server documentation for details.

Alternatively, you may consider using the EclipseLink persistence unit in conjunction with your database schema for auditing purposes.

7.5. About Connections

A connection is an object that provides access to a data source by way of the driver you configure your application to use. Relational projects use JDBC to connect to the data source; EIS projects use JCA. EclipseLink uses the interface `org.eclipse.persistence.internal.databaseaccess.Accessor` to wrap data source connections. This interface is accessible from certain events.

Typically, when using a server session, EclipseLink uses a different connection for both reading and writing. This lets you use nontransactional connections for reading and avoid maintaining connections when not required.

By default, an EclipseLink server session acquires connections lazily: that is, only during the commit operation of a persistence unit. Alternatively, you can configure EclipseLink to acquire a write connections at the time you acquire a client sessions.

Connections can be allocated from internal or external connection pools.

7.6. About Connection Pools

A **connection pool** is a service that creates and maintains a shared collection (pool) of data source connections on behalf of one or more clients. The connection pool provides a connection to a process on request, and returns the connection to the pool when the process is finished using it. When it is returned to the pool, the connection is available for other processes. Because establishing a connection to a data source can be time-consuming, 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.

You can configure your session to use internal connection pools provided by EclipseLink or external connection pools provided by a JDBC driver or Jakarta EE container.

You can use connection pools in your EclipseLink application for a variety of purposes, such as reading, writing, sequencing, and other application-specific functions.

This section describes the following types of connection pools:

- Internal Connection Pools
- External Connection Pools
- Default (Write) and Read Connection Pools
- Sequence Connection Pools
- Application-Specific Connection Pools

Internal Connection Pools

For non-Jakarta EE applications, you typically use *internal* connection pools. By default, EclipseLink sessions use internal connection pools.

Using internal connection pools, you can configure the default (write) and read connection pools. You can also create additional connection pools for object identity, or any other purpose.

Internal connection pools allow you to optimize the creation of read connections for applications that read data only to display it and only infrequently modify data. This 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.

External Connection Pools

For Jakarta EE applications, you typically use *external* connection pools. An external connection pool is a collection of reusable connections to a single data source provided by a JDBC driver or Jakarta EE container.

If you are using an external transaction controller (JTA), you must use external connection pools to integrate with the JTA.

Using external connection pools, you can use Java to configure the default (write) and read connection pools and create additional connection pools for object identity, or any other purpose.

External connection pools enable your EclipseLink application to do the following:

- Integrate into a Jakarta EE-enabled system.
- Integrate with JTA transactions (JTA transactions require a JTA-enabled data source).
- Leverage a shared connection pool in which multiple applications use the same data source.
- Use a data source configured and managed directly on the server.

Default (Write) and Read Connection Pools

A server session provides a read connection pool and a write connection pool. These could be different pools, or if you use external connection pooling, the same connection pool.

All read queries use connections from the read connection pool and all queries that write changes to the data source use connections from the write connection pool. You can configure attributes of the default (write) and read connection pools.

Whenever a new connection is established, EclipseLink uses the connection configuration you specify in your session's `DatasourceLogin`. Alternatively, when you use an external transaction controller, you can define a separate connection configuration for a read connection pool to avoid the additional overhead, if appropriate.

Use the `connection-pool.read` property to configure a read connection pool for non-transaction read queries. By default, EclipseLink does not use a separate read connection pool; the default pool is used for read queries. For more information, see `connection-pool.read` in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Sequence Connection Pools

An essential part of maintaining object identity is sequencing–managing the assignment of unique values to distinguish one instance from another. For more information, see About Cache Type and Size.

Sequencing involves reading and writing a special sequence resource maintained by your data source.

By default, EclipseLink includes sequence operations in a separate transaction. This avoids complications during the write transaction, which may lead to deadlocks over the sequence resource. However, when using an external transaction controller (such as a JTA data source or connection pool), EclipseLink cannot use a different transaction for sequencing. Use a sequence connection pool to configure a non-JTA transaction pool for sequencing. This is required only for table sequencing–not native sequencing.

In each server session, you can create one connection pool, called a sequence connection pool, that EclipseLink uses exclusively for sequencing. With a sequence connection pool, EclipseLink satisfies a request for a new object identifier outside of the transaction from which the request originates. This allows EclipseLink to immediately commit an update to the sequence resource, which avoids deadlocks.

If you use a sequence connection pool and the original transaction fails, the sequence operation does not roll back.

You should use a sequence connection pool, if the following applies:

- You use table sequencing (that is, non-native sequencing).
- You use external transaction controller (JTA).

You should not use a sequence connection pool, if the following applies:

- You do not use sequencing, or use the data source's native sequencing.
- You have configured the sequence table to avoid deadlocks.
- You use non-JTA data sources.

You can configure a sequence connection pool with the `eclipselink.connection-pool.sequence` persistence unit property. This property allows the connection pool to allocate generated IDs, and is required only for `TABLE` sequencing. By default, EclipseLink does not use a separate sequence connection pool; the default pool is used for sequencing. For more information, see `connection-pool.sequence` in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Application-Specific Connection Pools

When you use internal EclipseLink connection pools in a session, you can create one or more connection pools that you can use for any application purpose. These are called named connection pools, as you can give them any name you want and use them for any purpose.

Typically, use these named connection pools to provide pools of different security levels. For example, the "default" connection pool may only allow access to specific tables but the "admin" connection pool may allow access to all tables.

7.7. About Data Partitioning Policies

Data partitioning allows an application to scale its data across more than a single database machine. For more information on data partitioning, see "Using Data Partitioning to Scale Data" in *Solutions Guide for EclipseLink*.

Some databases support clustering the database across multiple machines. Oracle 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. For more information, see "Clustered Databases and Oracle RAC" in *Solutions Guide for EclipseLink*.

7.8. About Tenant Isolation

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 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 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 VPD Multi-Tenancy.

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.

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. 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.

For information on constructing table-per-tenant multi-tenancy, see "Using Table-Per-Tenant Multi-Tenancy" in *Solutions Guide for EclipseLink*.

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 using `@Multitenant` and `@TenantDiscriminatorColumn` annotations.

For more information on constructing VPD Multi-Tenancy, see "Using VPD Multi-Tenancy" in *Solutions Guide for EclipseLink*.

7.9. About Heterogeneous Batch Writing

The current release provides persistence unit properties to optimize transactions with multiple writes. The `eclipselink.jdbc.batch-writing` property configures the use of batch writing to optimize transactions with multiple writes. Batch writing allows multiple heterogeneous dynamic SQL statements to be sent to the database as a single execution, or multiple homogeneous parameterized SQL statements to be executed as a single batch execution.Note that not all JDBC drivers, or databases support batch writing.

The `eclipselink.jdbc.batch-writing.size` property configures the batch size used for batch writing. For parameterized batch writing this is the number of statements to batch, default 100. For dynamic batch writing, this is the size of the batched SQL buffer, default 32k.

The `eclipselink.jdbc.batch-writing` persistence property can also be used with query hints to configure if a modify query can be batched through batch writing. Some types of queries cannot be batched, such as DDL on some databases. Disabling batch writing will also allow the row count to be returned.

Chapter 8. Understanding Caching

This chapter introduces and describes 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 improves performance by holding recently read or written objects and accessing them in-memory to minimize database access, manage locking and isolation level, and manage object identity.

The entity caching annotations defined by EclipseLink are listed in "Caching Annotations" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

EclipseLink also provides a number of persistence unit properties that you can specify to configure the EclipseLink cache. These properties may compliment or provide an alternative to annotations. For a list of these properties, see "Caching" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

This chapter includes the following sections:

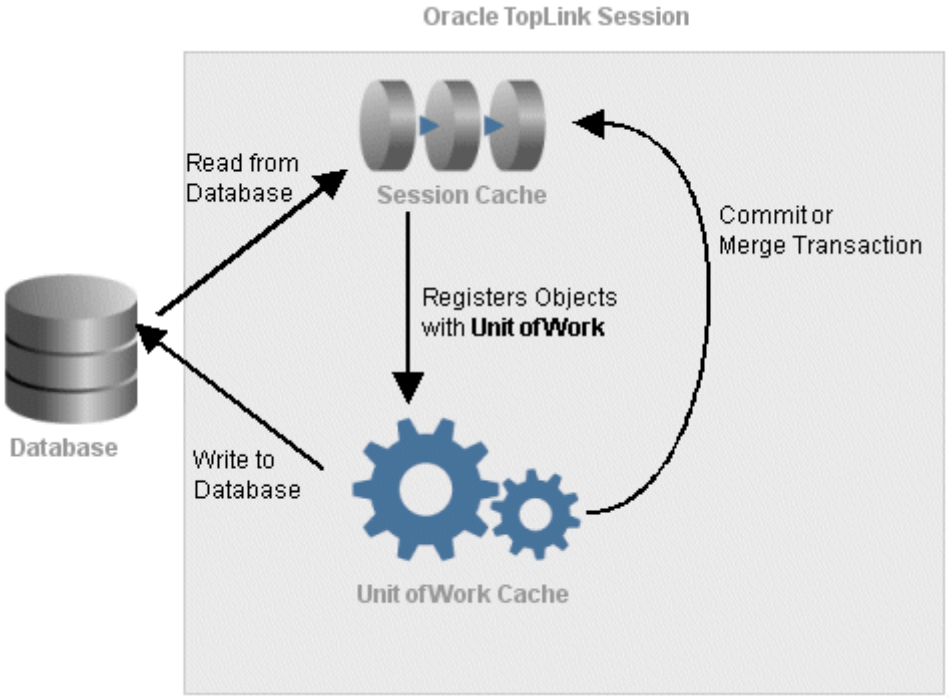
8.1. About Cache Architecture

EclipseLink uses two types of cache: the shared **persistence unit cache** (L2) maintains objects retrieved from and written to the data source; and the isolated **persistence context cache** (L1) holds objects while they participate in transactions. When a persistence context (entity manager) successfully commits to the data source, EclipseLink updates the persistence unit cache accordingly. Conceptually the persistence context cache is represented by the `EntityManager` and the persistence unit cache is represented by the `EntityManagerFactory`.

Internally, EclipseLink stores the persistence unit cache on a EclipseLink session, and the persistence context cache on a EclipseLink persistence unit. As Figure 8-1 shows, the persistence unit (session) cache and the persistence context (unit of work) cache work together with the data source connection to manage objects in a EclipseLink application.

Read requests from the database are sent to the persistence unit (session) cache in EclipseLink session. Write requests from the database are sent to the EclipseLink persistence context (unit of work) cache. The persistence unit (session) cache registers objects with the persistence context. During a commit or merge transaction, the persistence context cache refreshes the persistence unit cache. The object life cycle relies on these mechanisms.

Figure 8-1 Object Life Cycle and the EclipseLink Caches



Description of "Figure 8-1 Object Life Cycle and the EclipseLink Caches"

Persistence Unit Cache

The persistence unit cache is a shared cache (L2) that services clients attached to a given persistence unit. When you read objects from or write objects to the data source using an `EntityManager` object, EclipseLink saves a copy of the objects in the persistence unit's cache and makes them accessible to all other processes accessing the same persistence unit.

EclipseLink adds objects to the persistence unit cache from the following:

- The data store, when EclipseLink executes a read operation
- The persistence context cache, when a persistence context successfully commits a transaction

EclipseLink defines three cache isolation levels: Isolated, Shared, and Protected. For more information on these levels, see Shared, Isolated, Protected, Weak, and Read-only Caches.

There is a separate persistence unit cache for each unique persistence unit name. Although the cache is conceptually stored with the `EntityManagerFactory`, two factories with the same persistence unit name will share the same cache (and effectively be the same persistence unit instance). If the

same persistence unit is deployed in two separate applications in Jakarta EE, their full persistence unit name will normally still be unique, and they will use separate caches. Certain persistence unit properties, such as data-source, database URL, user, and tenant id can affect the unique name of the persistence unit, and result in separate persistence unit instances and separate caches. The `eclipselink.session.name` persistence unit property can be used to force two persistence units to resolve to the same instance and share a cache.

Persistence Context Cache

The persistence context cache is an isolated cache (L1) that services operations within an EntityManager. It maintains and isolates objects from the persistence unit cache, and writes changed or new objects to the persistence unit cache after the persistence context commits changes to the data source.

Only committed changes are merged into the shared persistence unit cache, flush or other operations do not affect the persistence unit cache until the transaction is committed.

The life-cycle for the persistence context cache differs between application managed, and container managed persistence contexts. The persistence context (unit of work) cache services operations within the persistence unit. It maintains and isolates objects from the persistence context (session) cache, and writes changed or new objects to the persistence context cache after the persistence unit commits changes to the data source.

Application Managed Persistence Contexts

An application managed persistence context is created by the application from an `EntityManagerFactory`. The application managed persistence context's cache will remain until the `EntityManager` is closed or `clear()` is called. It is important to keep application managed persistence units short lived, or to make use of `clear()` to avoid the persistence context cache from growing too big, or from becoming out of sync with the persistence unit cache and the database. Typically a separate `EntityManager` should be created for each transaction or request.

An extended persistence context has the same caching behavior as an application managed persistence context, even if it is managed by the container.

EclipseLink also supports a `WEAK` reference mode option for long lived persistence contexts, such as two-tier applications. See Weak Reference Mode.

Container Managed Persistence Contexts

A container managed persistence context is typically injected into a `SessionBean` or other managed object by a Jakarta EE container, or frameworks such as Spring. The container managed persistence context's cache will only remain for the duration of the transaction. Entities read in a transaction will become detached after the completion of the transaction and will require merging or editing in subsequent transactions.

EclipseLink supports accessing an entity's LAZY relationships after the persistence context has been closed.

Shared, Isolated, Protected, Weak, and Read-only Caches

EclipseLink defines three cache isolation levels. The cache isolation level defines how caching for an entity is performed by the persistence unit and the persistence context. The cache isolation levels can be set with the `isolation` attribute on the `@Cache` annotation. the possible values of the isolation attribute are:

- `isolated`—entities are only cached in the persistence context, not in the persistence unit. See Isolated Cache.
- `shared`—entities are cached both in the persistence context and persistence unit, read-only entities are shared and only cached in the persistence unit. See Shared Cache.
- `protected`—entities are cached both in the persistence context and persistence unit, read-only entities are isolated and cached in the persistence unit and persistence context. See Protected Cache.

Isolated Cache

The isolated cache (L1) is the cache stored in the persistence context. It is a transactional or user session based cache. Setting the cache isolation to `isolated` for an entity disables its shared cache. With an isolated cache all queries and find operations will access the database unless the object has already been read into the persistence context and refreshing is not used.

Use a isolated cache to do the following:

- avoid caching highly volatile data in the shared cache
- achieve serializable transaction isolation

Each persistence context owns an initially empty isolated cache. The persistence context's isolated cache is discarded when the persistence context is closed, or the `EntityManager.clear()` operation is used.

When you use an `EntityManager` to read an isolated entity, the `EntityManager` reads the entity directly from the database and stores it in the persistence context's isolated cache. When you read a read-only entity it is still stored in the isolated cache, but is not change tracked.

The persistence context can access the database using a connection pool or an exclusive connection. The persistence unit property `eclipselink.jdbc.exclusive-connection.mode` can be used to use an exclusive connection. Using an exclusive connection provides improved user-based security for reads and writes. Specific queries can also be configured to use the persistence context's exclusive connection.

If an `EntityManager` contains an exclusive connection, you must close the `EntityManager` when you are finished using it. We do not recommend relying on the finalizer to release the connection when the `EntityManager` is garbage-collected. If

> you are using a managed persistence context, then you do not need to close it.

Shared Cache

The shared cache (L2) is the cache stored in the persistence unit. It is a shared object cache for the entire persistence unit. Setting the cache isolation to `shared` for an entity enables its shared cache. With a shared cache queries and find operations will resolve against the shared cache unless refreshing is used.

Use a shared cache to do the following:

- improve performance by avoiding database access when finding or querying an entity by Id or index;
- improve performance by avoiding database access when accessing an entity's relationships;
- preserve object identity across persistence contexts for read-only entities.

When you use an `EntityManager` to find a shared entity, the `EntityManager` first checks the persistence unit's shared cache. If the entity is not in the persistence unit's shared cache, it will be read from the database and stored in the persistence unit's shared cache, a copy will also be stored in the persistence context's isolated cache. Any query not by Id, and not by an indexed attribute will first access the database. For each query result row, if the object is already in the shared cache, the shared object (with its relationships) will be used, otherwise a new object will be built from the row and put into the shared cache, and a copy will be put into the isolated cache. The isolated copy is always returned, unless read-only is used. For read-only the shared object is returned as the isolated copy is not required.

The size and memory usage of the shared cache depends on the entities cache type. attributes on the `@Cache` annotation can also be used to invalidate or clear the cache.

Protected Cache

The protected cache option allows for shared objects to reference isolated objects. Setting the cache isolation to `protected` for an entity enables its shared cache. The protected option is mostly the same as the shared option, except that protected entities can have relationships to isolated entities, whereas shared cannot.

Use a protected cache to do the following:

- improve performance by avoiding database access when finding or querying an entity by Id or index
- improve performance by avoiding database access when accessing an entity's relationships to shared entities
- ensure read-only entities are isolated to the persistence context
- allow relationships to isolated entities

Protected entities have the same life-cycle as shared entities, except for relationships, and read-only. Protected entities relationships to shared entities are cached in the shared cache, but their relationships to isolated entities are isolated and not cached in the shared cache. The `@Noncacheable` annotation can also be used to disable caching of a relationship to shared entities. Protected entities that are read-only are always copied into the isolated cache, but are not change tracked.

Weak Reference Mode

EclipseLink offers a specialized persistence context cache for long-lived persistence contexts. Normally it is best to keep persistence contexts short-lived, such as creating a new `EntityManager` per request, or per transaction. This is referred to as a stateless model. This ensures the persistence context does not become too big, causing memory and performance issues. It also ensures the objects cached in the persistence context do not become stale or out of sync with their committed state.

Some two-tier applications, or stateful models require long-lived persistence contexts. EclipseLink offers a special weak reference mode option for these types of applications. A weak reference mode maintains weak references to the objects in the persistence context. This allows the objects to garbage-collected if not referenced by the application. This helps prevent the persistence context from becoming too big, reducing memory usage and improving performance. Any new, removed or changed objects will be held with strong references until a commit occurs.

A weak reference mode can be configured through the `eclipselink.persistence-context.reference-mode` persistence unit property. The following options can be used:

- `HARD`—This is the default, weak references are not used. The persistence context will grow until cleared or closed.
- `WEAK`—Weak references are used. Unreferenced unchanged objects will be eligible for garbage collection. Objects that use deferred change tracking will not be eligible for garbage collection.
- `FORCE_WEAK`—Weak references are used. Unreferenced, unchanged objects will be eligible for garbage collection. Changed (but unreferenced) objects that use deferred change tracking will also be eligible for garbage collection, causing any changes to be lost.

Read-Only Entities

An entity can be configured as read-only using the `@ReadOnly` annotation or the `read-only` XML attribute. A read-only entity will not be tracked for changes and any updates will be ignored. Read-only entities cannot be persisted or removed. A read-only entity must not be modified, but EclipseLink does not currently enforce this. Modification to read-only objects can corrupt the persistence unit cache.

Queries can also be configured to return read-only objects using the `eclipselink.read-only` query hint.

A `shared` entity that is read-only will return the shared instance from queries. The same entity will be returned from all queries from all persistence contexts. Shared read-only entities will never be copied or isolated in the persistence context. This improves performance by avoiding the cost of

copying the object, and tracking the object for changes. This both reduces memory, reduces heap usage, and improves performance. Object identity is also maintained across the entire persistence unit for read-only entities, allowing the application to hold references to these shared objects.

An `isolated` or `protected` entity that is read-only will still have an isolated copy returned from the persistence context. This gives some improvement in performance and memory usage because it does not track the object for changes, but it is not as significant as `shared` entities.

8.2. About Cache Type and Size

EclipseLink provides several different cache types which have different memory requirements. The size of the cache (in number of cached objects) can also be configured. The cache type and size to use depends on the application, the possibility of stale data, the amount of memory available in the JVM and on the machine, the garbage collection cost, and the amount of data in the database.

The cache type of the shared object cache and its size can be configured with the `type` and `size` attributes of the `@Cache` annotation. In addition, the cache type for the query results cache can be configured with the `eclipselink.query-results-cache.type` persistence unit property. For more information, see the `@Cache` annotation and `eclipselink.query-results-cache.type` persistence unit property descriptions in the *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

By default, EclipseLink uses a `SOFT_WEAK` with an initial size of 100 objects. The cache size is not fixed, but just the initial size, EclipseLink will never eject an object from the cache until it has been garbage collected from memory. It will eject the object if the `CACHE` type is used, but this is not recommended. The cache size of the `SOFT_WEAK` and `HARD_WEAK` is also the size of the soft or hard sub-cache that can determine a minimum number of objects to hold in memory.

You can configure how object identity is managed on a class-by-class basis. The `ClassDescriptor` object provides the cache and identity map options described in Table 8-1.

Table 8-1 Cache and Identity Map Options

Option	Caching	Guaranteed Identity	Memory Use
FULL Cache Type	Yes	Yes	Very High
WEAK Cache Type	Yes	Yes	Low
SOFT Cache Type	Yes	Yes	High
SOFT_WEAK and HARD_WEAK Cache Type	Yes	Yes	Medium-high

There are two other options, `NONE`, and `CACHE`. These options are not recommend.

The value of the `type` attribute can be overridden with these persistence unit properties: `eclipselink.cache.type.<ENTITY>` and `eclipselink.cache.type.default`.

FULL Cache Type

This option provides full caching and guaranteed identity: objects are never flushed from memory unless they are deleted.

It caches all objects and does not remove them. Cache size doubles whenever the maximum size is reached. This method may be memory-intensive when many objects are read. Do not use this option on batch operations.

Oracle recommends using this identity map when the data set size is small and memory is in large supply.

WEAK Cache Type

This option only caches objects that have not been garbage collected. Any object still referenced by the application will still be cached.

The weak cache type uses less memory than full identity map but also does not provide a durable caching strategy across client/server transactions. Objects are available for garbage collection when the application no longer references them on the server side (that is, from within the server JVM).

SOFT Cache Type

This option is similar to the weak cache type, except that the cache uses soft references instead of weak references. Any object still referenced by the application will still be cached, and objects will only be removed from the cache when memory is low.

The soft identity map allows for optimal caching of the objects, while still allowing the JVM to garbage collect the objects if memory is low.

SOFT_WEAK and HARD_WEAK Cache Type

These options are similar to the weak cache except that they maintain a most frequently used sub-cache. The sub-cache uses soft or hard references to ensure that these objects are not garbage collected, or only garbage collected only if the JVM is low on memory.

The soft cache and hard cache provide more efficient memory use. They release objects as they are garbage-collected, except for a fixed number of most recently used objects. Note that weakly cached objects might be flushed if the transaction spans multiple client/server invocations. The size of the sub-cache is proportional to the size of the cache as specified by the `@Cache size` attribute. You should set the cache size to the number of objects you wish to hold in your transaction.

Oracle recommends using this cache in most circumstances as a means to control memory used by the cache.

NONE and CACHE

`NONE` and `CACHE` options do not preserve object identity and should only be used in very specific circumstances. `NONE` does not cache any objects. `CACHE` only caches a fixed number of objects in an `LRU` fashion. These cache types should only be used if there are no relationships to the objects.Oracle does not recommend using these options. To disable caching, set the cache isolation to `ISOLATED` instead.

Guidelines for Configuring the Cache and Identity Maps

Use the following guidelines when configuring your cache type:

- For objects with a long life span, use a `SOFT`, `SOFT_WEAK` or `HARD_WEAK` cache type. For more information on when to choose one or the other, see About the Internals of Weak, Soft, and Hard Cache Types..
- For objects with a short life span, use a `WEAK` cache type.
- For objects with a long life span, that have few instances, such as reference data, use a `FULL` cache type.

Use the `FULL` cache type only if the class has a small number of finite instances. Otherwise, a memory leak will occur.

- If caching is not required or desired, disable the shared cache by setting the cache isolation to `ISOLATED`.

Oracle does not recommend the use of `CACHE` and `NONE` cache types.

See About the Internals of Weak, Soft, and Hard Cache Types.

About the Internals of Weak, Soft, and Hard Cache Types

The `WEAK` and `SOFT` cache types use JVM weak and soft references to ensure that any object referenced by the application is held in the cache. Once the application releases its reference to the object, the JVM is free to garbage collection the objects. When a weak or a soft reference is garbage collected is determined by the JVM. In general, expect a weak reference to be garbage collected with each JVM garbage-collection operation.

The `SOFT_WEAK` and `HARD_WEAK` cache types contain the following two caches:

- Reference cache: implemented as a `LinkedList` that contains soft or hard references, respectively.
- Weak cache: implemented as a `Map` that contains weak references.

When you create a `SOFT_WEAK` or `HARD_WEAK` cache with a specified size, the reference cache

`LinkedList` is exactly this size. The weak cache `Map` has the size as its initial size: the weak cache will grow when more objects than the specified size are read in. Because EclipseLink does not control garbage collection, the JVM can reap the weakly held objects whenever it sees fit.

Because the reference cache is implemented as a `LinkedList`, new objects are added to the end of the list. Because of this, it is by nature a least recently used (LRU) cache: fixed size, object at the top of the list is deleted, provided the maximum size has been reached.

The `SOFT_WEAK` and `HARD_WEAK` are essentially the same type of cache. The `HARD_WEAK` was constructed to work around an issue with some JVMs.

If your application reaches a low system memory condition frequently enough, or if your platform's JVM treats weak and soft references the same, the objects in the reference cache may be garbage-collected so often that you will not benefit from the performance improvement provided by it. If this is the case, Oracle recommends that you use the `HARD_WEAK`. It is identical to the `SOFT_WEAK` except that it uses hard references in the reference cache. This guarantees that your application will benefit from the performance improvement provided by it.

When an object in a `HARD_WEAK` or `SOFT_WEAK` is pushed out of the reference cache, it gets put in the weak cache. Although it is still cached, EclipseLink cannot guarantee that it will be there for any length of time because the JVM can decide to garbage-collect weak references at anytime.

8.3. About Queries and the Cache

A query that is run against the shared persistence unit (session) cache is known as an **in-memory query**. Careful configuration of in-memory querying can improve performance.

By default, a query that looks for a single object based on primary key attempts to retrieve the required object from the cache first, and searches the data source only if the object is not in the cache. All other query types search the database first, by default. You can specify whether a given query runs against the in-memory cache, the database, or both.

About Query Cache Options and In-memory Querying

JPA defines standard query hints for configuring how a query interacts with the shared persistence unit cache (L2). EclipseLink also provides some additional query hints for configuring the cache usage. For information on JPA and EclipseLink query hints, see About Query Hints.

Entities can be accessed through JPA using either `find()` method or queries. The `find()` method will first check the persistence context cache (L1) for the Id, if the object is not found it will check the shared persistence unit cache (L2), if the object is still not found it will access the database. By default all queries will access the database, unless querying by Id or by cache indexed fields. Once the query retrieves the rows from the database, it will resolve each row with the cache. If the object is already in the cache, then the row will be discarded, and the object will be used. If the object is not in the shared cache, then it will be built from the row and put into the shared cache. A copy will also be put in the persistence context cache and returned as the query result.

This is the general process, but it differs if the transaction is dirty. If the transaction is dirty then the shared persistence unit cache will be ignored and objects will be built directly into the persistence

context cache.

A transaction is considered dirty in the following circumstances:

- A `flush()` has written changes to the database.
- A pessimistic lock query has been executed.
- An update or delete query has been executed.
- A native SQL query has been executed.
- This persistence unit property `eclipselink.transaction.join-existing` is used.
- The JDBC connection has been unwrapped from the EntityManager.
- The `UnitOfWork` API `beginEarlyTransaction` has been called.

Entities can also be configured to be isolated, or noncacheable, in which case they will never be placed in the shared cache (see "Shared, Isolated, Protected, Weak, and Read-only Caches").

8.4. About Handling Stale Data

Stale data is an artifact of caching, in which an object in the cache is not the most recent version committed to the data source. To avoid stale data, implement an appropriate cache locking strategy.

By default, EclipseLink optimizes concurrency to minimize cache locking during read or write operations. Use the default EclipseLink isolation level, unless you have a very specific reason to change it. For more information on isolation levels in EclipseLink, see Shared, Isolated, Protected, Weak, and Read-only Caches.

Cache locking regulates when processes read or write an object. Depending on how you configure it, cache locking determines whether a process can read or write an object that is in use within another process.

A well-managed cache makes your application more efficient. There are very few cases in which you turn the cache off entirely, because the cache reduces database access, and is an important part of managing object identity.

To make the most of your cache strategy and to minimize your application's exposure to stale data, Oracle recommends the following:

- Configuring a Locking Policy
- Configuring the Cache on a Per-Class Basis
- Forcing a Cache Refresh when Required on a Per-Query Basis
- Configuring Cache Invalidation
- Configuring Cache Coordination

Configuring a Locking Policy

Make sure you configure a locking policy so that you can prevent or at least identify when values have already changed on an object you are modifying. Typically, this is done using optimistic locking. EclipseLink offers several locking policies such as numeric version field, time-stamp version field, and some or all fields. Optimistic and pessimistic locking are described in the following sections.

Optimistic Locking

Oracle recommends using EclipseLink optimistic locking. With optimistic locking, all users have read access to the data. When a user attempts to write a change, the application checks to ensure the data has not changed since the user read the data. Use `@OptimisticLocking` to specify the type of optimistic locking EclipseLink should use when updating or deleting entities.

You can use version or field locking policies. Oracle recommends using version locking policies. 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.

For more information on the `OptimisticLocking` annotation and the types of locking you can use, see "@OptimisticLocking" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

For more information, see Optimistic Version Locking Policies and Optimistic Field Locking Policies.

Pessimistic Locking

With pessimistic locking, the first user who accesses the data with the purpose of updating it locks the data until completing the update. The disadvantage of this approach is that it may lead to reduced concurrency and deadlocks. Use the `eclipselink.pessimistic-lock` property to specify if EclipseLink uses pessimistic locking. For more information, see "eclipselink.pessimistic-lock" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Consider using pessimistic locking support at the query level. See Pessimistic Locking Policies.

Configuring the Cache on a Per-Class Basis

If other applications can modify the data used by a particular class, use a weaker style of cache for the class. For example, the `@Cache type` attribute values `WEAK` and `SOFT_WEAK` minimizes the length of time the cache maintains an object whose reference has been removed. For more information about cache types, see About Cache Type and Size.

Forcing a Cache Refresh when Required on a Per-Query Basis

Any query can include a flag that forces EclipseLink to go to the data source for the most recent version of selected objects and update the cache with this information. For more information, see About Explicit Query Refreshes. See also "Refreshing the Cache" in *Solutions Guide for EclipseLink*.

Configuring Cache Invalidation

You can configure any entity with an `expiry` that lets you specify either the number of milliseconds after which an entity instance should expire from the cache, or a time of day that all instances of the entity class should expire from the cache. Expiry is set on the `@Cache` annotation or `<cache>` XML element, and can be configured either with the `expiry` or with the `expiryTimOfDay` attribute. For more information, see "Setting Entity Caching Expiration" in *Solutions Guide for EclipseLink*.

Configuring Cache Coordination

If your application is primarily read-based and the changes are all being performed by the same Java application operating with multiple, distributed sessions, you may consider using the EclipseLink cache coordination feature. Although this will not prevent stale data, it should greatly minimize it. For more information, see About Cache Coordination and Clustering and Cache Coordination.

8.5. About Explicit Query Refreshes

Some distributed systems require only a small number of objects to be consistent across the servers in the system. Conversely, other systems require that several specific objects must always be guaranteed to be up-to-date, regardless of the cost. If you build such a system, you can explicitly refresh selected objects from the database at appropriate intervals, without incurring the full cost of distributed cache coordination.

To implement this type of strategy, do the following:

1. Configure a set of queries that refresh the required objects.
2. Establish an appropriate refresh policy.
3. Invoke the queries as required to refresh the objects.

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. For more information, see "Refreshing the Cache" in *Solutions Guide for EclipseLink*.

When you execute a query, if the required objects are in the cache, EclipseLink returns the cached objects without checking the database for a more recent version. This reduces the number of objects that EclipseLink must build from database results, and is optimal for noncoordinated cache

environments. However, this may not always be the best strategy for a coordinated cache environment.

To override this behavior, set the `alwaysRefresh` attribute to specify that the objects from the database always take precedence over objects in the cache. This updates the cached objects with the data from the database.

You can implement this type of refresh policy on each EclipseLink entity, or just on certain queries, depending upon the nature of the application.

8.6. About Cache Indexes

The EclipseLink cache is indexed by the entity's Id. This allows the `find()` operation, relationships, and queries by Id to obtain cache hits and avoid database access. The cache is not used by default for any non-Id query. All non-Id queries will access the database then resolve with the cache for each row returned in the result-set.

Applications tend to have other unique keys in their model in addition to their Id. This is quite common when a generated Id is used. The application frequently queries on these unique keys, and it is desirable to be able to obtain cache hits to avoid database access on these queries.

Cache indexes allow an in-memory index to be created in the EclipseLink cache to allow cache hits on non-Id fields. The cache index can be on a single field, or on a set of fields. The indexed fields can be updateable, and although they should be unique, this is not a requirement. Queries that contain the indexed fields will be able to obtain cache hits. Only single results can be obtained from indexed queries.

Cache indexes can be configured using the `@CacheIndex` and `@CacheIndexes` annotations and `<cache-index>` XML element. A `@CacheIndex` can be defined on the entity, or on an attribute to index the attribute. Indexes defined on the entity must define the `columnNames` used for the index. An index can be configured to be re-indexed when the object is updated using the updateable attribute.

It is still possible to cache query results for non-indexed queries using the query result cache. For more information, see About Query Results Cache.

8.7. Database Event Notification and Oracle CQN

Some databases and database products allow events to be raised from the database when rows are updated or deleted.

EclipseLink supports an API to allow the database to notify EclipseLink of database changes, so the changed objects can be invalidated in the EclipseLink shared cache. This allows a shared cache to be used, and stale data to be avoided, even if other applications access the same data in the database. EclipseLink supports integration with the Oracle Database feature for Database Change Notification CQN. A custom `DatabaseEventListener` may be provided for other databases and products that support database events.

There are also other solutions to caching in a shared environment, including:

- Disable the shared cache (through setting `@Cacheable(false)`, or `@Cache(isolation=ISOLATED)`).
- Only cache read-only objects.
- Set a cache invalidation timeout to reduce stale data.
- Use refreshing on objects/queries when fresh data is required.
- Use optimistic locking to ensure write consistency (writes on stale data will fail, and will automatically invalidate the cache).

The JPA Cache API and the EclipseLink `JpaCache` API can also be used directly to invalidate objects in the shared cache by the application. EclipseLink cache coordination could also be used to send invalidation messages to a cluster of EclipseLink persistence units.

Database events can reduce the chance of an application getting stale data, but do not eliminate the possibility. Optimistic locking should still be used to ensure data integrity. Even in a single server application stale data is still possible within a persistence context unless pessimistic locking is used. Optimistic (or pessimistic) locking is always required to ensure data integrity in any multi-user system.

Oracle Continuous Query Notification

The Oracle database released a Continuous Query Notification (CQN) feature in the 10.2 release. CQN allows for database events to be raised when the rows in a table are modified. The JDBC API for CQN was not complete until 11.2, so 11.2 is required for EclipseLink's integration.

EclipseLink CQN support is enabled by the `OracleChangeNotificationListener` listener which integrates with Oracle JDBC to received database change events. Use the `eclipselink.cache.database-event-listener` property to configure the full class name of the listener.

By default all tables in the persistence unit are registered for change notification, but this can be configured using the `databaseChangeNotificationType` attribute of the `@Cache` annotation to selectively disable change notification for certain classes.

Oracle CQN uses the `ROWID` to inform of row level changes. This requires EclipseLink to include the `ROWID` in all queries for a CQN enabled class. EclipseLink must also select the object's `ROWID` after an insert operation. EclipseLink must maintain a cache index on the `ROWID`, in addition to the object's Id. EclipseLink also selects the database transaction Id once each transaction to avoid invalidating the cache on the server that is processing the transaction.

EclipseLink's CQN integration has the following limitations:

- Changes to an object's secondary tables will not trigger it to be invalidate unless a version 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 a version is used and updated in the primary table.

8.8. About Query Results Cache

The EclipseLink query results cache allows the results of named queries to be cached, similar to

how objects are cached.

By default in EclipseLink all queries access the database, unless they are by Id, or by cache-indexed fields. The resulting rows will still be resolved with the cache, and further queries for relationships will be avoided if the object is cached, but the original query will always access the database. EclipseLink does have options for querying the cache, but these options are not used by default, as EclipseLink cannot assume that all of the objects in the database are in the cache. The query results cache allows for non-indexed and result list queries to still benefit from caching.

The query results cache is indexed by the name of the query, and the parameters of the query. Only named queries can have their results cached, dynamic queries cannot use the query results cache. As well, if you modify a named query before execution, such as setting hints or properties, then it cannot use the cached results.

The query results cache does not pick up committed changes from the application as the object cache does. It should only be used to cache read-only objects, or should use an invalidation policy to avoid caching stale results. Committed changes to the objects in the result set will still be picked up, but changes that affect the results set (such as new or changed objects that should be added/removed from the result set) will not be picked up.

The query results cache supports a fixed size, cache type, and invalidation options.

8.9. About Cache Coordination

The need to maintain up-to-date data for all applications is a key design challenge for building a distributed application. The difficulty of this increases as the number of servers within an environment increases. EclipseLink provides a distributed cache coordination feature that ensures data in distributed applications remains current.

Cache coordination reduces the number of optimistic lock exceptions encountered in a distributed architecture, and decreases the number of failed or repeated transactions in an application. However, cache coordination in no way eliminates the need for an effective locking policy. To effectively ensure working with up-to-date data, cache coordination must be used with optimistic or pessimistic locking. Oracle recommends that you use cache coordination with an optimistic locking policy.

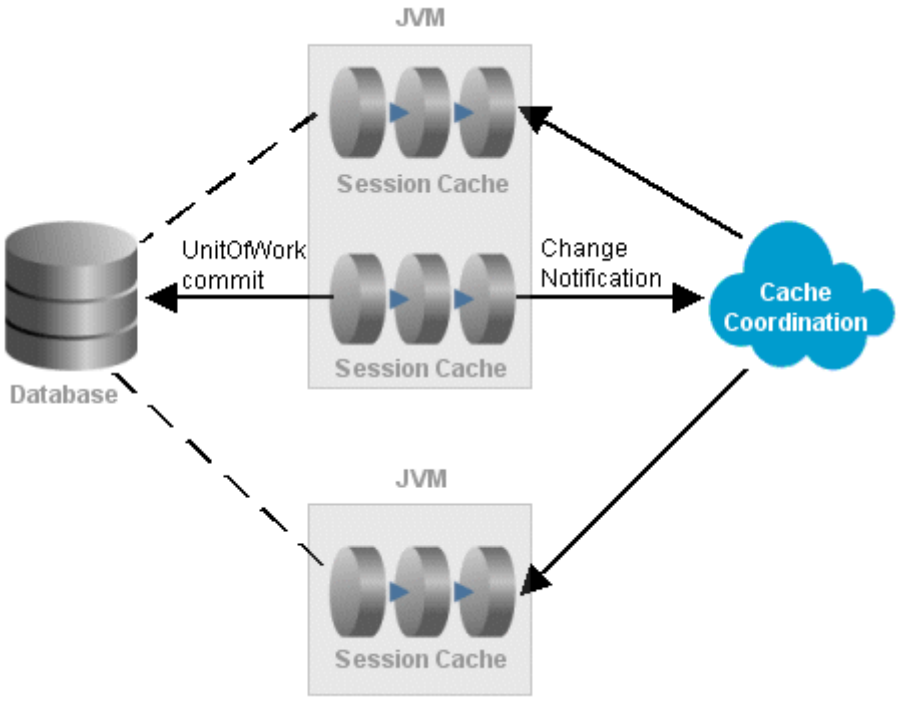
Tune the EclipseLink cache for each class to help eliminate the need for distributed cache coordination. Always tune these settings before implementing cache coordination. For more information, see "Monitoring and Optimizing EclipseLink-Enabled Applications" in *Solutions Guide for EclipseLink*.

You can use cache invalidation to improve cache coordination efficiency. For more information, see "Setting Entity Caching Expiration" in *Solutions Guide for EclipseLink*.

As Figure 8-2 shows, cache coordination is a session feature that allows multiple, possibly distributed, instances of a session to broadcast object changes among each other so that each session’s cache is either kept up-to-date or notified that the cache must update an object from the data source the next time it is read.

You cannot use isolated client sessions with cache coordination. For more information, see Shared, Isolated, Protected, Weak, and Read-only Caches.

Figure 8-2 Coordinated Persistence Unit (Session) Caches



Description of "Figure 8-2 Coordinated Persistence Unit (Session) Caches"

When sessions are distributed, that is, when an application contains multiple sessions (in the same JVM, in multiple JVMs, possibly on different servers), as long as the servers hosting the sessions are interconnected on the network, sessions can participate in cache coordination. Coordinated cache types that require discovery services also require the servers to support User Datagram Protocol (UDP) communication and multicast configuration. For more information, see Coordinating JMS and RMI Caches.

When to Use Cache Coordination

Cache coordination can enhance performance and reduce the likelihood of stale data for applications that have the following characteristics:

- Changes are all being performed by the same Java application operating with multiple, distributed sessions
- Primarily read-based
- Regularly requests and updates the same objects

To maximize performance, avoid cache coordination for applications that do not have these characteristics.

For other options to reduce the likelihood of stale data, see About Handling Stale Data.

8.10. Clustering and Cache Coordination

An application cluster is a set of middle tier server machines or VMs servicing requests for a single application, or set of applications. Multiple servers are used to increase the scalability of the application and/or to provide fault tolerance and high availability. Typically the same application will be deployed to all of the servers in the cluster and application requests will be load balanced across the set of servers. The application cluster will access a single database, or a database cluster. An application cluster may allow new servers to be added to increase scalability, and for servers to be removed such as for updates and servicing.

Application clusters can consist of Jakarta EE servers, Web containers, or Java server applications.

EclipseLink can function in any clustered environment. The main issue in a clustered environment is utilizing a shared persistence unit (L2) cache. If you are using a shared cache (enabled by default in EclipseLink projects), then each server will maintain its own cache, and each caches data can get out of synchronization with the other servers and the database.

EclipseLink provides cache coordination in a clustered environment to ensure the servers caches are synchronized.

There are also many other solutions to caching in a clustered environment, including:

- Disable the shared cache (through setting `@Cacheable(false)`, or `@Cache(isolation=ISOLATED)`).
- Use only cache read-only objects.
- Set a cache invalidation timeout to reduce stale data.
- Use refreshing on objects/queries when fresh data is required.
- Use optimistic locking to ensure write consistency (writes on stale data will fail, and will automatically invalidate the cache).
- Use database events to invalidate changed data in the cache (such as EclipseLink's support for Oracle Query Change Notification).

Cache coordination enables a set of persistence units deployed to different servers in the cluster (or on the same server) to synchronize their changes. Cache coordination works by each persistence unit on each server in the cluster being able to broadcast notification of transactional object changes to the other persistence units in the cluster. EclipseLink supports cache coordination over RMI and JMS. The cache coordination framework is also extensible so other options could be developed.

By default, EclipseLink optimizes concurrency to minimize cache locking during read or write operations. Use the default EclipseLink transaction isolation configuration unless you have a very specific reason to change it.

Cache coordination works by broadcasting changes for each transaction to the other servers in the cluster. Each other server will receive the change notification, and either invalidate the changed objects in their cache, or update the cached objects state with the changes. Cache coordination

occurs after the database commit, so only committed changes are broadcast.

Cache coordination greatly reduces to chance of an application getting stale data, but does not eliminate the possibility. Optimistic locking should still be used to ensure data integrity. Even in a single server application stale data is still possible within a persistence context unless pessimistic locking is used. Optimistic (or pessimistic) locking is always required to ensure data integrity in any multi-user system.

For more information about cache coordination, including cache synchronization, see "Using Cache Coordination" in *Solutions Guide for EclipseLink*.

Coordinating JMS and RMI Caches

For a JMS coordinated caches, when a particular session's coordinated cache starts, it uses its JNDI naming service information to locate and create a connection to the JMS server. The coordinated cache is ready when all participating sessions are connected to the same topic on the same JMS server. At this point, sessions can start sending and receiving object change messages. You can then configure all sessions that are participating in the same coordinated cache with the same JMS and JNDI naming service information.

For an RMI coordinated cache, when a particular session's coordinated cache starts, the session binds its connection in its naming service (either an RMI registry or JNDI), creates an announcement message (that includes its own naming service information), and broadcasts the announcement to its multicast group. When a session that belongs to the same multicast group receives this announcement, it uses the naming service information in the announcement message to establish bidirectional connections with the newly announced session's coordinated cache. The coordinated cache is ready when all participating sessions are interconnected in this way, at which point sessions can start sending and receiving object change messages. You can then configure each session with naming information that identifies the host on which the session is deployed.

For more information on configuring JMS and RMI cache coordination, see "Configuring JMS Cache Coordination Using Persistence Properties" and "Configuring RMI Cache Coordination Using Persistence Properties" in *Solutions Guide for EclipseLink*.

Coordinating Custom Caches

You can define your own custom solutions for coordinated caches by using the classes in the EclipseLink `org.eclipse.persistence.sessions.coordination` package.

8.11. Clustering and Cache Consistency

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.

To ensure cache consistency you perform tasks such as disabling entity caching, refreshing the cache, setting entity expiration, and setting optimistic locking on the cache. For more information on these topics, see "Task 1: Configure Cache Consistency" in *Solutions Guide for EclipseLink*.

8.12. Cache Interceptors

EclipseLink provides a very functional, performant and integrated cache. However, you can integrate third-party external caches by using the EclipseLink `CacheInterceptor` annotation and API.

Chapter 9. Understanding Queries

This chapter describes how EclipseLink enables you to create, read, update, and delete persistent objects or data using queries in both Jakarta EE and non-Jakarta EE applications for both relational and nonrelational data sources.

This chapter includes the following sections:

9.1. Query Concepts

In general, querying a data source means performing an action on or interacting with the contents of the data source. To do this, you must be able to perform the following:

- Define an action in a syntax native to the data source being queried.
- Apply the action in a controlled fashion.
- Manage the results returned by the action (if any).

You must also consider how the query affects the EclipseLink cache.

This section introduces query concepts unique to EclipseLink, including the following:

Call Objects

The `Call` object encapsulates an operation or action on a data source. The EclipseLink API provides a variety of `Call` types such as structured query language (SQL), Java Persistence Query Language

(JPQL), and Extensible Markup Language (XML).

You can execute a `Call` directly or in the context of the EclipseLink `DatabaseQuery` object.

DatabaseQuery Objects

A `DatabaseQuery` object is an abstraction that associates additional customization and optimization options with the action encapsulated by a `Call`. By separating these options from the `Call`, EclipseLink can provide sophisticated query capabilities across all `Call` types.

Data-Level and Object-Level Queries

Queries can be defined for objects or data, as follows:

- **Object-level** queries are object-specific and return data as objects in your domain model. They are the preferred type of query for mapped data. By far, object-level `DatabaseQuery` queries are the most common query used in EclipseLink.
- **Data-level** queries are used to query database tables directly, and are an appropriate way to work with unmapped data.

Summary Queries

While data-level queries return raw data and object-level queries return objects in your domain model, summary queries return data about objects. EclipseLink provides partial object queries to return a set of objects with only specific attributes populated, and report queries to return summarized (or rolled-up) data for specific attributes of a set of objects.

Descriptor Query Manager

In addition to storing named queries applicable to a particular class, you can also use the `DescriptorQueryManager` to override the default action that EclipseLink defines for common data source operations.

Query Keys

A query key is a schema-independent alias for a database field name. Using a query key, you can refer to a field using a schema-independent alias. In relational projects only, EclipseLink automatically creates query keys for all mapped attributes. The name of the query key is the name of the class attribute specified in your object model.

You can configure query keys in a class descriptor or interface descriptor. You can use query keys in expressions and to query variable one-to-one mappings.

By default, EclipseLink creates query keys for all mapped attributes, but in some scenarios you may find it beneficial to add your own.

9.2. About JPQL Queries

The Java Persistence Query Language (JPQL) is the query language defined by JPA. JPQL is similar to SQL, but operates on objects, attributes and relationships instead of tables and columns. JPQL can be used for reading (`SELECT`), as well as bulk updates (`UPDATE`) and deletes (`DELETE`). JPQL can be used in a `NamedQuery` (through annotations or XML) or in dynamic queries using the `EntityManager createQuery()` API.

The disadvantage of JPQL is that dynamic queries require performing string concatenations to build queries dynamically from web forms or dynamic content. JPQL is also not checked until runtime, making typographical errors more common. These disadvantages are reduced by using the query Criteria API, described About the Criteria API.

EclipseLink Extensions to JPQL

EclipseLink supports all of the statements and clauses described in "Query Language" in the JPA Specification, including `SELECT` queries, update and delete statements, `WHERE` clauses, literal values, and database functions. For more information, see the JPA Specification.

`http://jcp.org/en/jsr/detail?id=338`

EclipseLink provides many extensions to the standard JPA JPQL. These extensions provide access to additional database features many of which are part of the SQL standard, provide access to native database features and functions, and provide access to EclipseLink specific features.

EclipseLink's JPQL extensions include:

- Less restrictions than JPQL, allows sub-selects and functions within operations such as `LIKE`, `IN`, `ORDER BY`, constructors, functions etc.
- Allow != in place of <>
- `FUNCTION` operation to call database specific functions
- `TREAT` operation to downcast related entities with inheritance
- `OPERATOR` operation to call EclipseLink database independent functions
- `SQL` operation to mix `SQL` with JPQL
- `CAST` and `EXTRACT` functions
- `REGEXP` function for regular expression querying
- Usage of sub-selects in the `SELECT` and `FROM` clause
- `ON` clause support for defining `JOIN` and `LEFT JOIN` conditions

- Joins between independent entities
- Usage of an alias on a `JOIN FETCH`
- `COLUMN` operation to allow querying on nonmapped columns
- `TABLE` operation to allow querying on non mapped tables
- `UNION`, `INTERSECT`, `EXCEPT` support
- Usage of object variables in =, <>, `IN`, `IS NULL`, and `ORDER BY`

For descriptions of these extensions, see "EclipseLink Query Language" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

EclipseLink Special Operators in JPQL

EclipseLink defines several special JPQL operators that allow performing database operations that are not possible in basic JPQL. These include:

- `COLUMN`
- `FUNCTION`
- `OPERATOR`
- `SQL`

For descriptions of these operators, see "Special Operators" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

9.3. About SQL Query Language

Using EclipseLink, you can express a query using the following query languages:

- SQL Queries
- EclipseLink Expressions (see Chapter 10, "Understanding EclipseLink Expressions")
- Stored Procedures

SQL is the most common query language for applications that use a relational database data source. In most cases, you can compose a query directly in a given query language or, preferably, you can construct a `DatabaseQuery` with an appropriate `Call` and specify selection criteria using an `Expression` object. Although composing a query directly in SQL appears to be the simplest approach (and for simple operations or operations on unmapped data, it is), using the `DatabaseQuery` approach offers the compelling advantage of confining your query to your domain object model and avoiding dependence on data source schema implementation details.

Oracle recommends that you compose your queries using `Expression`.

You can execute custom SQL directly using `Session` methods `executeSelectingCall` and `executeNonSelectingCall`, or you can construct a `DatabaseQuery` with an appropriate `Call`.

EclipseLink provides a variety of SQL `Call` objects for use with stored procedures and, with Oracle Database, stored functions. EclipseLink also supports PLSQL call for Oracle stored procedures with PLSQL data types.

Stored Procedures

As described in the "Stored Procedures" section of the JPA specification (http://jcp.org/en/jsr/detail?id=338), native SQL allows you to use named stored procedures either dynamically or specified by the `NamedStoredProcedureQuery` annotation. If you use annotations, the stored procedure must exist in the database. The annotation allows you to specify the types of all parameters to the stored procedure, their corresponding parameter modes, and the mapping of the result sets.

Metadata must be provided for all parameters by using the `StoredProcedureParameter` annotation. Parameters must be specified in the order in which they occur in the parameter list of the stored procedure. If parameter names are used, the parameter name is used to bind the parameter value and to extract the output value (if the parameter is an `INOUT` or `OUT` parameter).

If the stored procedure is not defined using metadata, then parameter and result set information must be provided dynamically.

EclipseLink Extensions to Stored Procedures

EclipseLink defines annotation extensions that allow the use of PLSQL stored procedures (such as `@NamedPLSQLStoredProcedureQuery`) and stored functions (such as `@NamedPLSQLStoredFunctionQuery`). The PLSQL annotations allow you to use complex PLSQL types such as `RECORD` and `TABLE`, that are not accessible from JDBC. The annotations contain attributes for specifying the function (or procedure) name, the return value of the stored function, any query hints, the parameters for the stored function, and the name of the `SQLResultMapping`.

Parameters for the stored function (or procedure) are specified with the `@PLSQLParameter` annotation. The `@PLSQLRecord` annotation defines a database PLSQL `RECORD` type for use within PLSQL procedures.

EclipseLink also defines annotation extensions that allow the use of non-PLSQL stored procedures (such as `@NamedStoredProcedureQuery`) and stored functions (such as `@NamedStoredFunctionQuery`).

For a list of the EclipseLink extensions for stored procedures and links to their descriptions, see "Stored Procedure and Function Annotations" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

9.4. About the Criteria API

The Java Persistence Criteria API is used to define dynamic queries through the construction of object-based query definition objects, rather than use of the string-based approach of JPQL. The Criteria API allows dynamic queries to be built programmatically offering better integration with the Java language than a string-based 4th GL approach.

The Criteria API has two modes, the type-restricted mode, and the non-typed mode. The type-restricted mode uses a set of JPA metamodel generated classes to define the query-able attributes of

a class. The non-typed mode uses strings to reference attributes of a class.

The Criteria API is only for dynamic queries, and cannot be used in metadata or named queries. Criteria queries are dynamic queries and do not perform as well as static named queries, or even dynamic parametrized JPQL which benefit from EclipseLink's parse cache.

For more information, see Chapter 6 "Criteria API" in the JPA Specification.

http://jcp.org/en/jsr/detail?id=338

`CriteriaBuilder` is the main interface into the Criteria API. A `CriteriaBuilder` is obtained from an `EntityManager` or an `EntityManagerFactory` using the `getCriteriaBuilder()` API. `CriteriaBuilder` is used to construct `CriteriaQuery` objects and their expressions. The Criteria API currently only supports select queries.

`CriteriaQuery` defines a database select query. A `CriteriaQuery` models all of the clauses of a JPQL select query. Elements from one `CriteriaQuery` cannot be used in other `CriteriaQuerys`. A `CriteriaQuery` is used with the `EntityManager createQuery()` API to create a JPA Query.

The `where` clause is normally the main part of the query as it defines the conditions (predicates) that filter what is returned. The `where` clause is defined using the `where` API on `CriteriaQuery` with any `Predicate` objects. A `Predicate` is obtained using a comparison operation, or a logical operation on `CriteriaBuilder`. The `isNull`, `isNotNull`, and `in` operations can also be called on `Expression` objects. The `not` operation can also be called on `Predicate` objects.

Subqueries can be used in the Criteria API in the `select`, `where`, `order`, `group by`, or `having` clauses. A subquery is created from a `CriteriaQuery` using the `subquery` operation. Most `subquery` usage restricts the subquery to returning a single result and value, unless used with the `CriteriaBuilder exists`, `all`, `any`, or `some` operations, or with an `in` operation.

Parameters can be defined using the `parameter` API on `CriteriaBuilder`. JPA defines named parameters, and positional parameters. For named parameters the parameter type and name are specified. For positional parameters only the parameter type is specified. Positional parameters start at position `1` not `0`.

Several database functions are supported by the Criteria API. All supported functions are defined on `CriteriaBuilder`. Some functions may not be supported by some databases, if they are not SQL compliant, and offer no equivalent function.

The Criteria API defines several special operations that are not database functions, but have special meaning in JPA. Some of these operations are defined on `CriteriaBuilder` and some are on specific Expression interfaces.

JPA defines a meta-model that can be used at runtime to query information about the ORM mapping metadata. The meta-model includes the list of mapped attributes for a class, and their mapping types and cardinality. The meta-model can be used with the Criteria API in place of using strings to reference the class attributes.

JPA defines a set of "_" classes ("_MyEntity.java", for example) that are to be generated by the JPA provider, or IDE, that give compile time access to the meta-model. This allows typed static variables to be used in the Criteria API. This can reduce the occurrence of typos, or invalid queries in

application code, by catching query issues at compile time, instead of during testing. It does however add complexity to the development process, as the meta-model static class needs to be generated, and be part of the development cycle.

A Tuple defines a multi-select query result. Normally an object array is returned by JPA multi-select queries, but an object array is not a very useful data structure. A Tuple is a map-like structure that allows the results to be retrieved by name or index.

EclipseLink Extensions to the Criteria API

EclipseLink's Criteria API support has fewer restrictions than specified by JPA. In general, sub-queries and object path expressions are allowed in most places, including:

- Sub-queries in the select, group by, and order clauses;
- Sub-query usage with functions;
- in usage with object path expressions;
- Order by usage with object path expressions.

EclipseLink's Criteria API support is built on top of EclipseLink native `Expression` API. EclipseLink provides the `JpaCriteriaBuilder` interface to allow the conversion of native `Expression` objects to and from JPA `Expression` objects. This allows the EclipseLink native `Expression` API to be mixed with the JPA Criteria API.

The EclipseLink native `Expression` API provides the following additional functionality:

- Additional database functions (over 80 database functions are supported)
- Usage of custom `ExpressionOperators`
- Embedding of SQL within an `Expression` query
- Usage of sub-selects in the from clause
- `ON` clause support
- Access to unmapped columns and tables
- Historical querying

EclipseLink `Expressions` can be combined with EclipseLink `DatabaseQuerys` to provide additional functionality:

- Unions, intersect and except clauses;
- Hierarchical connect by clauses;
- Batch fetching.

9.5. About Native SQL Queries

JPA allows SQL to be used for querying entity objects, or data. SQL queries are not translated, and passed directly to the database. SQL queries can be used for advanced queries that require database specific syntax, or by users who are more comfortable in the SQL language than JPQL or

Java.

SQL queries are created from the `EntityManager` using the `createNativeQuery` API or via named queries. A Query object is returned and executed the same as any other JPA query. An SQL query can be created for an entity class, or return an object array of data. If returning entities, the SQL query must return the column names that the entity's mappings expect, or an `SqlResultSetMapping` can be used. An `SqlResultSetMapping` allows the SQL result set to be mapped to an entity, or set of entities and data.

SQL queries can be used to execute SQL or DML (Data Manipulation Language) statements. For SQL queries that return results, `getSingleResult` or `getResultList` can be used. For SQL queries that do not return results, `executeUpdate` must be used. `executeUpdate` can only be used within a transaction. SQL queries can be used to execute database operations and some stored procedures and functions. Stored procedures that return output parameters, or certain complex stored procedures, cannot be executed with SQL queries.

Parameters to SQL queries are delimited using the question mark (`?`) character. Only indexed parameters are supported, named parameters are not supported. The index can be used in the delimiter, such as `?1`. Parameter values are set on the Query using the `setParameter` API. Indexed parameters start at the index 1 not 0.

Native SQL queries can be defined as named queries in annotations or XML using the `NamedNativeQuery` annotation or `<named-native-query>` XML element. Named native SQL queries are executed the same as any named query.

An `SqlResultSetMapping` can be used to map the results of an SQL query to an entity if the result column names do not match what the entity mappings expect. It can also be used to return multiple entities, or entities and data from a single SQL query. `EntityResult` and `FieldResult` are used to map the SQL query result column to the entity attribute. `ColumnResult` can be used to add a data element to the result.

`SqlResultSetMappings` are defined through annotations or XML using the `@SqlResultSetMapping` annotation or `<sql-result-set-mapping>` XML element. They are referenced from native SQL queries by name.

EclipseLink Extensions to Native Query Support

EclipseLink expressions let you specify query search criteria based on your domain object model. When you execute the query, EclipseLink translates these search criteria into the appropriate query language for your platform.

The EclipseLink API provides the following two public classes to support expressions:

- The `Expression` class represents an expression that can be anything from a simple constant to a complex clause with boolean logic. You can manipulate, group, and integrate expressions.
- The `ExpressionBuilder` class is the factory for constructing new expressions.

You can specify a selection criterion as an `Expression` with `DatabaseQuery` method

`setSelectionCriteria`, and in a finder that takes an `Expression`.

For more information about using EclipseLink expressions, see Chapter 10, "Understanding EclipseLink Expressions".

9.6. About Query Hints

You can use a query hint to customize or optimize a JPA query. The `NamedQuery` annotation is used to specify a named query in the Java Persistence query language. This annotation contains a `hints` element that can be used to specify query properties and hints. For more information on this annotation, see "NamedQuery Annotation" in the JPA Specification.

`http://jcp.org/en/jsr/detail?id=338`

The definitions of query hints are vendor-specific. The following sections describe JPA query hints and EclipseLink query hints:

- JPA Cache Query Hints
- EclipseLink Extensions to Cache Query Hints

JPA Cache Query Hints

The JPA query hints allow for queries or the `find()` operation to bypass, or refresh the shared cache. JPA cache query hints can be set on named or dynamic queries, or set in the properties map passed to the `find()` operation.

JPA 2.0 defines the following query hint properties to configure a queries interaction with the shared cache:

- `jakarta.persistence.cache.retrieveMode`
- `jakarta.persistence.cache.storeMode`

EclipseLink Extensions to Cache Query Hints

The EclipseLink cache query hints allow for queries or the `find()` operation to interact with the cache is the following ways:

- Bypass the cache check and force accessing the database, but still resolve with the cache.
- Refresh the cache from the database results.
- Bypass the cache and persistence unit and return detached objects.
- Bypass the persistence context and return read-only objects.
- Allow queries that use Id fields, and other fields to obtain cache hits.
- Query the cache first, and only access the database if the object is not found.
- Only query the cache, and avoid accessing the database.
- Conform a query with non-flushed changes in a persistence context.

Queries that access the cache have the following restrictions:

- Sub-selects are not supported.
- Certain database functions are not supported.
- Queries must return a single set of objects.
- Grouping is not supported.
- Uninstantiated lazy relationships may not be able to be queried.

All EclipseLink query hints are defined in the `QueryHints` class in the `org.eclipse.persistence.config` package. When you set a hint, you can set the value using the `public static final` field in the appropriate configuration class in `org.eclipse.persistence.config` package, including the following:

- `HintValues`
- `CacheUsage`
- `PessimisticLock`
- `QueryType`

You can specify EclipseLink query hints (JPA query extensions) either by using the `@QueryHint` annotation, by including the hints in the `orm.xml` or `eclipselink-orm.xml` files. or by using the `setHint()` method when executing a named or dynamic query (JPQL or Criteria).

Query settings and query hints that affect the generated SQL are not supported with SQL queries. Unsupported query hints include:

- `batch`
- `history.as-of`
- `inheritance.outer-join`
- `sql.hint`
- `join-fetch`—`join-fetch` is supported, but requires that the SQL selects all of the joined columns.
- `fetch-group`—`fetch-group` is supported, but requires that the SQL selects all of the fetched columns.
- `pessimistic-lock`—`pessimistic-lock` is supported, but requires that the SQL locks the result rows.

For descriptions of these extensions, see "EclipseLink Query Language" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

9.7. About Query Casting

Use query casting to query across attributes in subclasses when using JPA or ORM. This feature is available in JPQL, EclipseLink Expressions, and Criteria API.

Starting with JPA 2.0, it is possible to limit the results or a query to those of a specific subclass. For

example, the expression framework provides `Expression.type(Class)`.

In JPQL, downcasting is accomplished in the `FROM` clause, using `TREAT...AS` in the `JOIN` clause.

The JPA `Criteria` API includes the casting operator `Expression.as(type)`. This expression does a simple cast that allows matching of types within the generics.

Calling a cast on a `JOIN` node permanently alters that node. For example, in the example above, after calling `join.as(LargeProject.class)`, the join refers to a `LargeProject`.

EclipseLink Expression Support for Downcasting

EclipseLink extends the `Criteria` API to allow a cast using `Expression.as(type)`. The as method checks the hierarchy; and if type is a subclass of the type for the expression that is being called on, a cast is implemented.

The `Expression.as(Class)` can also be used for downcasting. The behavior of using `Expression.as(Class)` is as follows:

- An exception is thrown at query execution time if the class that is cast to is not a subclass of the class of the query key being cast.
- Casts are only allowed on `ObjectExpressions` (`QueryKeyExpression` and `ExpressionBuilder`). The parent expression of a cast must be an `ObjectExpression`.
- Casts use the same outer join settings as the `ObjectExpression` they modify.
- Casts modify their parent expression. As a result, when using a cast with a parallel expression, you must use a new instance of the parent expression.
- Casting is not supported for `TablePerClass` inheritance.
- It is prudent to do a check for type in a query that does a cast.
- EclipseLink automatically appends type information for cases where the cast results in a single type; but for classes in the middle of a hierarchy, no type information is appended to the SQL.

9.8. About Oracle Extensions for Queries

When you use EclipseLink with Oracle Database, you can make use of the following Oracle-specific query features from within your EclipseLink applications:

- Query Hints
- Hierarchical Queries
- Flashback Queries
- Stored Functions

Query Hints

Oracle lets you specify SQL query additions called hints that can influence how the database server

SQL optimizer works. This lets you influence decisions usually reserved for the optimizer. You use hints to specify things such as join order for a join statement, or the optimization approach for a SQL call.

You specify hints using the EclipseLink `DatabaseQuery` method `setHintString`.

For more information, see the performance tuning guide for your database.

Hierarchical Queries

Oracle Database Hierarchical Queries mechanism lets you select database rows based on hierarchical order. For example, you can design a query that reads the row of a given employee, followed by the rows of people the employee manages, followed by their managed employees, and so on.

You specify a hierarchical query clause using the `setHierarchicalQueryClause` method which appears in the EclipseLink `DatabaseQuery` subclass `ReadAllQuery`.

Flashback Queries

When using EclipseLink with Oracle9*i* Database (or later), you can acquire a special historical session where all objects are read as of a past time, and then you can express read queries depending on how your objects are changing over time. For more information, see "Using Oracle Flashback Technology" in *Oracle Database Advanced Application Developer's Guide*.

Stored Functions

A stored function is an Oracle Database mechanism that provides all the capabilities of a stored procedure in addition to returning a value. provides a number of annotations for working with stored functions as well as stored procedures. For a list of the EclipseLink annotation extensions for stored functions and procedures and links to their descriptions, see "Stored Procedure and Function Annotations" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

Chapter 10. Understanding EclipseLink Expressions

this chapter describes how to use the EclipseLink expressions framework to specify query search criteria based on your domain object model.

This chapter includes the following sections:

10.1. About the Expression Framework

The EclipseLink expression framework provides methods through the following classes:

- The `Expression` class provides most general functions, such as `toUpperCase`.
- The `ExpressionMath` class supplies mathematical methods.

This division of functionality enables EclipseLink expressions to provide similar mathematical functionality to the `java.lang.Math` class, but keeps both the `Expression` and `ExpressionMath` classes from becoming unnecessarily complex.

Expressions offer the following advantages over SQL when you access a database:

- Expressions are easier to maintain because the database is abstracted.
- Changes to descriptors or database tables do not affect the querying structures in the application.
- Expressions enhance readability by standardizing the `Query` interface so that it looks similar to traditional Java calling conventions.
- Expressions allow read queries to transparently query between two classes that share a relationship. If these classes are stored in multiple tables in the database, EclipseLink automatically generates the appropriate join statements to return information from both tables.
- Expressions simplify complex operations.

10.2. About Expression Components

A simple expression usually consists of the following three parts:

- The *attribute*, which represents a mapped attribute or query key of the persistent class
- The *operator*, which is an expression method that implements boolean logic, such as `GreaterThan`, `Equal`, or `Like`
- The *constant* or *comparison*, which refers to the value used to select the object

In the following code fragment, the attribute is `lastName`, the operator is `equal` and the constant is

the string “Smith”. The `expressionBuilder` substitutes for the object or objects to be read from the database. In this example, `expressionBuilder` represents employees.

```
expressionBuilder.get("lastName").equal("Smith");
```

You can use the following components when constructing an `Expression`:

- Boolean Logic
- Database Functions and Operators
- Oracle XMLType Functions
- Platform and User-Defined Functions
- Expressions for One-to-One and Aggregate Object Relationships
- Expressions for Joining and Complex Relationships

Boolean Logic

Expressions use standard boolean operators, such as `AND`, `OR`, and `NOT`, and you can combine multiple expressions to form more complex expressions.

Database Functions and Operators

EclipseLink supports many database functions using standard operator names that are translated to different databases. EclipseLink operators are supported on any database that has an equivalent function (or set of functions). For more information and a list of all supported functions and operators see "OPERATOR" and "FUNCTION" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*

EclipseLink expressions support a variety of database functions, which are described in the `Expression` class. Database functions let you define more flexible queries. You can use these functions in either a report query using a `SELECT` clause, or with comparisons in a query’s selection criteria using a `WHERE` clause.

Some functions may be database platform-specific.

Operators are relational operations that compare two values. EclipseLink expression operators are described in the `ExpressionOperator` class.

Mathematical functions are available through the `ExpressionMath` class. Mathematical function support in expressions is similar to the support provided by the Java class `java.lang.Math`.

Oracle XMLType Functions

You can use the following operators when constructing queries against data mapped to Oracle Database `XMLType` column:

- `extract`—Takes an XPath string and returns an XMLType which corresponds to the part of the original document that matches the XPath.
- `extractValue`—Takes an XPath string and returns either a numerical or string value based on the contents of the node pointed to by the XPath.
- `existsNode`—Takes an XPath expression and returns the number of nodes that match the XPath.
- `getStringVal`—Gets the string representation of an `XMLType` object.
- `getNumberVal`—Gets the numerical representation of an `XMLType` object.
- `isFragment`—Evaluates to 0 if the XML is a well formed document. Evaluates to 1 if the document is a fragment.

Platform and User-Defined Functions

You can use the `Expression` method `getFunction` to access database functions that EclipseLink does not support directly. The `Expression` API includes additional forms of the `getFunction` method that allow you to specify arguments. You can also create your own custom functions. For more information, see *Java API Reference for EclipseLink*.

Expressions for One-to-One and Aggregate Object Relationships

Expressions can include an attribute that has a one-to-one relationship with another persistent class. A one-to-one relationship translates naturally into an SQL join that returns a single row.

Expressions for Joining and Complex Relationships

You can query against complex relationships, such as one-to-many, many-to-many, direct collection, and aggregate collection relationships. Expressions for these types of relationships are more complex to build, because the relationships do not map directly to joins that yield a single row per object.

This section describes the following:

- About Joins
- Using EclipseLink Expression API for Joins

About Joins

A **join** is a relational database query that combines rows from two or more tables. Relational databases perform a join whenever multiple tables appear in the query's `FROM` clause. The query's select list can select any columns from any of these tables.

An inner join (sometimes called a "simple join") is a join of two or more tables that returns only those rows that satisfy the join condition.

An outer join extends the result of an inner join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition. Outer joins can be categorized as left or right:

- A query that performs a left outer join of tables A and B returns all rows from A. For all rows in A that have no matching rows in B, the database returns null for any select list expressions containing columns of B.
- A query that performs a right outer join of tables A and B returns all rows from B. For all rows in B that have no matching rows in A, the database returns null for any select list expressions containing columns of A.

When you query with a join expression, EclipseLink can use joins to check values from other objects or other tables that represent parts of the same object. Although this works well under most circumstances, it can cause problems when you query against a one-to-one relationship, in which one side of the relationship is not present.

For example, `Employee` objects may have an `Address` object, but if the `Address` is unknown, it is `null` at the object level and has a null foreign key at the database level. When you attempt a read that traverses the relationship, missing objects cause the query to return unexpected results. Consider the following expression:

```
(emp.get("firstName").equal("Steve")).or(emp.get("address").
get("city").equal("Ottawa"))
```

In this case, employees with no address do not appear in the result set, regardless of their first name. Although not obvious at the object level, this behavior is fundamental to the nature of relational databases.

Outer joins rectify this problem in the databases that support them. In this example, the use of an outer join provides the expected result: all employees named Steve appear in the result set, even if their address is unknown.

To implement an outer join, use `Expression` method `getAllowingNull`, rather than `get`, and `Expression` method `anyOfAllowingNone`, rather than `anyOf`.

For example:

```
(emp.get("firstName").equal("Steve")).or(
emp.getAllowingNull("address").get("city").equal("Ottawa"))
```

Support and syntax for outer joins vary widely between databases and database drivers. EclipseLink supports outer joins for most databases.

Using EclipseLink Expression API for Joins

You can use joins anywhere expressions are used, including: selection-criteria, ordering, report queries, partial objects, one-to-one relational mappings, and join reading.

Use the Expression API shown in Table 10-1 to configure inner and outer join expressions.

Table 10-1 Expression API for Joins

Expression API	Type of Join	Type of Mapping
`get`	inner	one-to-one
`getAllowingNull`	outer	one-to-one
`anyOf`	inner	one-to-many, many-to-many
`anyOfAllowingNone`	outer	one-to-many, many-to-many

To query across a one-to-many or many-to-many relationship, use the `anyOf` operation. As its name suggests, this operation supports queries that return all items on the "many" side of the relationship that satisfy the query criteria.

Chapter 11. Understanding Non-relational Data Sources

This chapter describes how to set up your JPA applications to work with a non-relational data source. There are many types of non-relational data sources. These include document databases, key-value stores, and various other non-standard databases, such as MongoDB, Cassandra, and Google BigTable. This chapter focuses on the NoSQL data source. NoSQL is a classification of database systems that do not support the SQL standard. EclipseLink supports persistence of Java objects to NoSQL databases through the Jakarta Persistence API (JPA). EclipseLink native API is also supported with NoSQL databases.

This chapter includes the following sections:

11.1. NoSQL Platform Concepts

NoSQL is a classification of database systems that do not support the SQL standard.The NoSQL classification can be expanded to include Enterprise Information Systems (EIS) including application databases, legacy databases, messaging systems, and transaction processing monitors, such as IMS, VSAM, and ADABASE.

EclipseLink's NoSQL support includes:

- MongoDB
- Oracle NoSQL
- XML files
- JMS
- Oracle AQ

A complete description of EclipseLink support for NoSQL is described in Non-SQL Standard Database Support: NoSQL.

NoSQL and EIS data-sources have a Java Connector Architecture (JCA) resource adapter that supports the Java Connector Architecture Common Client Interface (JCA CCI).

There are many different ways to access NoSQL and EIS data-sources. Many NoSQL data-sources provide a Java API. For EIS data-sources, there are many third party and custom Java adapters. These APIs are normally non-standard, and low-level, similar to JDBC. EclipseLink NoSQL support is

built on top of such APIs, and offers the rich, high-level and standard JPA API.

Some NoSQL data-sources support the JDBC API and a subset of the SQL language. Many third-party vendors provide JDBC drivers for EIS data-sources. EclipseLink regular JPA support can be used with any compliant JDBC driver. So, if JDBC access is an option, EclipseLink's NoSQL support is not required, as EclipseLink standard JPA support can be used.

Some NoSQL data-sources may support JCA. JCA is a Java Enterprise Edition API that allows connecting to more generic systems than JDBC. JCA is composed of two parts, a resource adapter layer, and the Common Client Interface (CCI). EclipseLink NoSQL and EIS support is based on the JCA CCI. For MongoDB, Oracle NoSQL, XML files, JMS, and Oracle AQ, EclipseLink provides the JCA adapter and EclipseLink `EISPlatform` and `ConnectionSpec` classes. Third party JCA adapters can also be used with EclipseLink as long as they support the CCI. There are third party JCA vendors, such as Attunity, that support various EIS data-sources such as IMS, VSAM and ADABASE.

11.2. About NoSQL Persistence Units

NoSQL persistence units are configured the same as JPA persistence units. The `persistence.xml` file is used to define the persistence unit. NoSQL persistence units can be application managed, JTA managed, injected, or created through Persistence the same as regular JPA persistence units. NoSQL persistence units do have some specific persistence unit properties that are required, and have some limitations.

NoSQL defines the following persistence unit properties:

- `eclipselink.nosql.connection-spec`
- `eclipselink.nosql.connection-factory`
- `eclipselink.nosql.property`
- `eclipselink.target-database`—this is used to set the NoSQL platform class, or use `org.eclipse.persistence.eis.EISPlatform` for a generic platform.

For more information on these properties, see "Persistence Property Extensions Reference"in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*.

NoSQL persistence units have the following restrictions:

- `<jta-data-source>`, `<non-jta-data-source>`—these elements are not supported, as they refer to JDBC `DataSources`.
- JTA—JTA-managed persistence units are supported, but XA transactions may not be provided unless the NoSQL JCA resource adapter supports JTA.
- `javax.jdbc`, `eclipselink.jdbc`—JDBC-specific properties are not supported as NoSQL does not use JDBC.

Persistence Unit Properties for NoSQL Platforms

To use a NoSQL platform you must set both the `eclipselink.nosql.connection-spec` to the connection spec class name and the `eclipselink.target-database` to the platform class name. Each

NoSQL platform also supports platform-specific properties that can be set using `eclipselink.nosql.property`. For more information on values for MongoDB, Oracle NoSQL, XML, JMS, and Oracle AQ, see "@NoSql" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLink*. See also Non-SQL Standard Database Support: NoSQL.

11.3. About JPA Applications on the NoSQL Platform

Mapping to NoSQL data is configured through the EclipseLink `@NoSql` annotation, and `<no-sql>` XML element. `@NoSql` defines the class as mapping to non-relational data. `@NoSql` can be specified with `@Entity` or `@Embeddable` classes.

The `@NoSql` annotation defines a `dataType` and a `dataFormat` attribute. The `dataType` attribute is the name for the entity's structure; the meaning of the `dataType` is dependent on the NoSQL platform. For MongoDB, it is the collection name that the JSON documents are stored to. For Oracle NoSQL the `dataType` is the first part of the major key value. For the XML file adapter it is the file name.

The `dataFormat` attribute specifies the type of structure the data is stored as. The `dataFormat` attribute is defined by the `DataFormatType enum`.

For examples of configuring an application with the `@NoSql` annotation, see "@NoSql" in *Jakarta Persistence API (JPA) Extensions Reference for EclipseLinkk*

Mapping Restrictions on JPA Annotations

NoSQL supports most JPA annotations and others have different restrictions than mapping relational data.

Supported mapping annotations:

- `@Entity`—defines a root level object in the NoSQL data-store.
- `@Embeddable`—defines an object embedded in another object's data structure.
- `@Basic`, `@Temporal`, `@Enumerated`, `@Lob`
- `@Convert`, `@Converter`, `@TypeConverter`, `@ObjectTypeConverter`
- `@Access`, `@Transient`, `@Mutable`
- `@Id`, `@EmbeddedId`
- `@GeneratedValue`, `@UuidGenerator`
- `@Version`—is supported, but dependent on the NoSQL data-source to validate version write conflicts.
- `@Embedded`—defines a reference that will be embedded in the parent's data structure as a nested structure.
- `@ElementCollection`—defines a collection of values or embeddables that will be embedded in the parent's data structure as a list of nested structures.
- `@OneToOne`, `@ManyToOne`—define a relationship to another root level object stored as a foreign key in the source object's data structure.

- `@OneToMany`, `@ManyToMany`—define a relationship to a collection of other root level object stored as a list of foreign keys in the source object's data structure.
- `@Inheritance`, `@MappedSuperclass`, `@ClassExtractor`
- `@Cacheable`, `@Cache`, `@ReadOnly`, `@Noncacheable`
- `@NamedQuery`—is supported on NoSQL data-sources that support querying.
- `@NamedNativeQuery`—is supported on NoSQL data-sources that support native querying. The query language is not SQL, but specific to the NoSQL data-store.
- `@EntityListeners`, `@PrePersist`, `@PreUpdate`, `@PreRemove`, `@PreLoad`, `@PostPersist`, `@PostUpdate`, `@PostRemove`, `@PostLoad`
- `@Customizer`

Unsupported mapping annotations:

- `@Table`, `@SecondaryTable`—are not supported, as objects are not mapped to tables, it is replaced by the dataType on the `@NoSql` annotation.
- `@Column`—`@Field` should be used, as data is not stored in table columns, however `@Column` is still allowed, but just the name will be used.
- `@JoinColumn`—is not supported; it is replaced by `@JoinField`.
- `@JoinTable`—is not required or supported; OneToManys and ManyToManys are stored as collections of Ids embedded in the source object's data structure.
- `@CollectionTable`—is not required or supported; ElementCollections are embedded in the parent object's data structure.
- `@MapKeyColumn`, `@MapKeyClass`, `@MapKeyJoinColumn`—are not currently supported.
- `@OrderBy`, `@OrderColumn`—are not normally required or supported, as order is normally maintained by the object's data structure.
- `@SequenceGenerator`, `@TableGenerator`—are not directly supported.
- `@AttributeOverride`, `@AssociationOverride`—are supported with inheritance, but are not supported or required with embedded relationships as embedded objects are nested in their parent object's data structure, not flattened as in the case of relational data.
- `@JoinFetch`, `@BatchFetch`—are not supported.

11.4. About Mapping NoSQL Objects

NoSQL maps objects to structured data such as XML or JSON. NoSQL supports embedded data, embedded collections, and all of the existing JPA mapping annotations.

To map NoSQL objects, you must define IDs, mapping, embedded objects, relationships, and locking. For more information, see "Implementing the Solution" in *Solutions Guide for EclipseLink*.

11.5. About Queries and the NoSQL Platform

Whether querying is supported in NoSQL depends on the NoSQL platform you are using. Some

NoSQL data-sources may support dynamic querying through their own query language, others may not support querying at all. The following types of queries are supported by NoSQL.

- JPQL Queries
- Native Queries
- Interaction Queries

For more information on support for these queries, see "Defining Queries" in *Solutions Guide for EclipseLink*.

11.6. About Transactions and the NoSQL Platform

The JPA transaction API is supported with NoSQL data-sources. Some NoSQL data-sources might not support transactions, so the level of transaction support is dependent on the NoSQL platform. JTA persistence units and transactions are also supported, but unless the NoSQL adapter is integrated with JTA, no XA or transaction support will be available.

If the NoSQL data-source does not support transactions, then any database change such as `flush()` will be committed directly to the database, and `rollback()` will not have any affect. A commit operation that fails will not roll back any successful changes written before the error. JPA normally does not write to the database until commit or `flush()` are called, so there will still be some level of transaction support offered by the persistence context.

JPA operations `persist()`, `merge()`, and `remove()` are supported.

- MongoDB—Transactions are not supported.
- Oracle NoSQL—Transactions are not supported.

Appendix A: Database and Application Server Support

This appendix describes the database platforms and application servers supported by EclipseLink.

This appendix includes the following sections:

A.1. Database Support

EclipseLink supports any relational database that is compliant with SQL and has a compliant JDBC driver. EclipseLink has extended support for several database platforms. The extended support mainly consists of providing native sequencing support, schema creation, and certain database functions.

The databases in Table A-1 are supported. The Java classes are in the `org.eclipse.persistence.platform.database` package and are described in *Java API Reference for EclipseLink*.

Table A-1 Supported Database Platforms

Database	Java Class	Features
Apache Derby	`org.eclipse.persistence.platform.database.DerbyPlatform`	Provides Derby-specific behavior.
Attunity	`org.eclipse.persistence.platform.database.AttunityPlatform`	Platform class that works with Attunity's Connect JDBC driver.
dBASE	`org.eclipse.persistence.platform.database.DBasePlatform` ``	Provides dBASE-specific behavior, including: • Writes `Time` and `Timestamp` as strings (dBASE does not support `Time`s or `Timestamp`s)
Firebird	`org.eclipse.persistence.platform.database.FirebirdPlatform`	Provides Firebird-specific behavior.
H2	`org.eclipse.persistence.platform.database.H2Platform`	Provides H2-specific behavior.

HyperSQL DataBase (HSQL)	`org.eclipse.persistence.platform.database.HSQLPlatform`	Provides HSQL-specific behavior. Supports HSQL functionality as of 1.8.1. Features include: • DDL creation • `IDENTITY` sequencing • `SEQUENCE` objects • Functions • Pagination
IBM Cloudscape	`org.eclipse.persistence.platform.database.CloudscapePlatform`	Provides CloudScape DBMS-specific behavior.
IBM DB2 Mainframe	`org.eclipse.persistence.platform.database.DB2MainframePlatform`	Provides DB2 Mainframe-specific behavior. This provides for some additional compatibility in certain DB2 versions on OS390. Features include: • Specialized `CONCAT` syntax
IBM DB2	`org.eclipse.persistence.platform.database.DB2Platform`	Provides DB2-specific behavior, including: • Schema creation • Native SQL for `byte[]`, `Date`, `Time`, and `Timestamp` • Table qualified names. • Stored procedures • Temporary tables • Casting • Database functions • Identity sequencing • `SEQUENCE` sequencing
IBM Informix	`org.eclipse.persistence.platform.database.InformixPlatform`	Provides Informix-specific behavior, including: • Types for schema creation. • Native sequencing using `@SERIAL`.
MariaDB	`org.eclipse.persistence.platform.database.MariaDBPlatform`	Provides MariaDB-specific behavior.

Microsoft Access	`org.eclipse.persistence.platform.database.AccessPlatformPlatform`	Provides Microsoft Access-specific behavior.
Microsoft SQLServer	`org.eclipse.persistence.platform.database.SQLServerPlatform`	Provides Microsoft SQL Server-specific behavior, including: • Native SQL for `byte[]`, `Date`, `Time`, and `Timestamp`. • Native sequencing using `@IDENTITY`.
MySQL	`org.eclipse.persistence.platform.database.MySQLPlatform`	Provides MySQL-specific behavior, including: • Native SQL for `Date`, `Time`, and `Timestamp` • Native sequencing • Mapping of class types to database types for the schema framework • Pessimistic locking • Platform specific operators

Oracle	`org.eclipse.persistence.platform.database.OraclePlatform`	Provides Oracle Database-specific behavior, including: • `LOB` • `NChar` • `XMLType` • `TIMESTAMP` (`TZ`, `LTZ`) • `JSON` • Native batch writing • Structured object-relational data-types • PLSQL datatypes and stored procedures • VPD, RAC, proxy authentication • XDK XML parser • Hierarchical selects (Select by prior) • Returning clause • Flashback history and queries • Stored procedures, output parameters and output cursors • Stored functions • Oracle AQ
Oracle JavaDB	`org.eclipse.persistence.platform.database.JavaDBPlatform`	Allows the use of `JavaDBPlatform` as a synonym for `DerbyPlatform`.
Oracle TimesTen	`org.eclipse.persistence.platform.database.TimesTenPlatform`	Provides Oracle TimesTen database-specific behavior.
Oracle TimesTen7	`org.eclipse.persistence.platform.database.TimesTen7Platform`	Provides Oracle TimesTen 7 database-specific behavior.
PervasivePlatform	`org.eclipse.persistence.platform.database.PervasivePlatform`	Provides Pervasive PSQL-specific behavior.
PointBase	`org.eclipse.persistence.platform.database.PointBasePlatform`	Provides PointBase database-specific behavior.

PostgreSQL	`org.eclipse.persistence.platform.database.PostgreSQLPlatform`	Provides PostgreSQL database-specific behavior, including: • Native SQL for `Date`, `Time`, and `Timestamp` • Native sequencing • Mapping of class types to database types for the schema framework • Pessimistic locking • Platform specific operators • `LIMIT`/`OFFSET` query syntax for select statements See also PostgreSQL: http://wiki.eclipse.org/EclipseLink/FAQ/JPA/PostgreSQL
PostgreSQL 10	`org.eclipse.persistence.platform.database.PostgreSQL10Platform`	Provides PostgreSQL database-specific behavior, plus: * `JSON` datatype support
SAP MaxDB	`org.eclipse.persistence.platform.database.MaxDBPlatform`	Provides MaxDB database-specific behavior.
SAP SyBase SQLAnywhere	`org.eclipse.persistence.platform.database.SQLAnywherePlatform`	Provides SQL Anywhere-specific behavior.
Sybase	`org.eclipse.persistence.platform.database.SybasePlatform`	Provides Sybase-specific behavior, including: • Native SQL for `byte[]`, `Date`, `Time`, and `Timestamp` • Native sequencing using `@IDENTITY`

Fujitsu Symfoware	`org.eclipse.persistence.platform.database.SymfowarePlatform`	Provides Symfoware-specific behavior, including: • DDL Generation • Outer Join • Subquery (with limitations) • Stored Procedure Calls • Stored Procedure Generation • Native Sequences/Identifier fields • JPA Bulk Update/Delete (with limitations) • Batch Reading • Batch Writing • Pessimistic Locking (with limitations) • First Result/Limit (with limitations) • Expression Framework (with limitations) • Delimiters • Auto Detection

It also possible to extend EclipseLink to add extended support for additional platforms. There are also several user-contributed platforms in the EclipseLink incubator project. See Platform Incubator:

`http://wiki.eclipse.org/EclipseLink/Development/Incubator/Platform`

A.2. Application Server Support

EclipseLink can be used with any Jakarta EE application server that meets the software requirements through the EclipseLink API.

Table A-2 lists the application servers for which EclipseLink provides integration support. The classes listed in the table are concrete subclasses of the EclipseLink `org.eclipse.persistence.platform.server.ServerPlatformBase` class, which is responsible for representing server-specific behavior. These classes determine the following behavior for the server:

- Which external transaction controller to use
- Whether to enable JTA (external transaction control)

- How to register or unregister for runtime services (JMX or MBean)
- Whether to enable runtime services
- How to launch container Threads

For more information on the Java classes listed in the table, see *Java API Reference for EclipseLink*.

Table A-2 Supported Application Servers

Server Name	Java Classes
Open Liberty Application Server	• `org.eclipse.persistence.platform.server.was.WebSphere_6_1_Platform` • `org.eclipse.persistence.platform.server.was.WebSphere_7_Platform` • `org.eclipse.persistence.platform.server.was.WebSpherePlatform` • `org.eclipse.persistence.platform.server.was.WebSphere_Liberty_Platform`
WildFly Application Server	• `org.eclipse.persistence.platform.server.jboss.JBossPlatform`
Oracle WebLogic Server	• `org.eclipse.persistence.platform.server.wls.WebLogic_10_Platform` • `org.eclipse.persistence.platform.server.wls.WebLogic_9_Platform` • `org.eclipse.persistence.platform.server.wls.WebLogicPlatform`
GlassFish Application Server	• `org.eclipse.persistence.platform.server.glassfish.GlassfishPlatform`

EclipseLink MOXy as the JAXB Provider for Application Servers

EclipseLink MOXy is integrated into the GlassFish and WebLogic application servers as the JAXB provider. For more information, see the following links:

- GlassFish Server: `http://blog.bdoughan.com/2012/02/glassfish-312-is-full-of-moxy.html`
- WebLogic Server: `http://blog.bdoughan.com/2011/12/eclipselink-moxy-is-jaxb-provider-in.html`

A.3. Non-SQL Standard Database Support: NoSQL

EclipseLink JPA can be used with NoSQL databases. A Java class can be mapped to a NoSQL datasource using the `@NoSQL` annotation or `<no-sql>` XML element.

EclipseLink also provides JPA access to EIS (Enterprise Information Systems) such as legacy

databases and systems (CICS, ADA, VSAM, IMS, MQ, AQ).

EclipseLink's NoSQL support allows complex hierarchical data to be mapped, including XML, indexed, and hierarchical mapped data such as JSON data. CRUD operations, embedded objects and collections, inheritance, and relationships are supported. A subset of JPQL and the Criteria API are supported, dependent on the NoSQL database's query support.

It is also possible to add support for other NoSQL data-sources by defining your own `EISPlatform` subclass and JCA adapter. There are also several user-contributed platforms in the EclipseLink incubator project. See "Platform Incubator" at this URL:

`http://wiki.eclipse.org/EclipseLink/Development/Incubator/Platform`

Table A-3 lists the NoSQL and EIS data-sources that are supported by EclipseLink. For more information on the Java classes listed in the table, see *Java API Reference for EclipseLink*.

Table A-3 Supported NoSQL and EIS Platforms

Data-source	Java Class	Features
MongoDB	`org.eclipse.persistence.nosql.adapters.mongo.MongoPlatform`	Provides MongoDB support including: • `MAPPED` JSON data • JPQL and Criteria queries • Native queries • hints for `READ_PREFERENCE`, `WRITE_CONCERN`, `OPTIONS`, `SKIP`, `LIMIT`, `BATCH_SIZE`
Oracle NoSQL	`org.eclipse.persistence.nosql.adapters.nosql.OracleNoSQLPlatform`	Provides Oracle NoSQL support including: • `MAPPED` key/value data • XML data • `find()` and `SELECT` all queries • hints for `CONSISTENCY`, `DURABILITY`, `TIMEOUT`, `VERSION`

XML files	`org.eclipse.persistence.eis.adapters.xmlfile.XMLFilePlatform`	Provides support for persistence to XML file including: • XML data • `find()` and `SELECT` all queries • XPath interactions
JMS	`org.eclipse.persistence.eis.adapters.jms.JMSPlatform`	Provides support for persistence through JMS messaging: • XML data • `send`/`receive` operations
Oracle AQ	`org.eclipse.persistence.eis.adapters.aq.AQPlatform`	Provides support for persistence through Oracle AQ messaging: • XML data • `enqueue`/`dequeue` operations