EclipseLink/UserGuide/JPA/Print Version

From Eclipsepedia

Contents

[hide]

Introduction to Java Persistence API

Related Topics

This section introduces concepts of Java Persistence API and provides general information on it.

What Is Java Persistence API?

The Java Persistence API (JPA) is a lightweight framework for Java persistence (see Persisting Objects) based on Plain Old Java Object (POJO). JPA is a part of EJB 3.0 specification. JPA 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. You can use this API to create, remove and query across lightweight Java objects within both an EJB 3.0-compliant container and a standard Java SE 5 environment.

For more information, see the following:

- Considering JPA Entity Architecture
- JSR 220 EJB 3.0 with JPA 1.0 specifications

What Do You Need to Develop with JPA

To start developing with JPA, you need the following:

- Relational Database
- Domain Model Classes
- persistence.xml File
- Object Relational Mapping Metadata
- Persistence Provider
- Persistence Application Code

Relational Database

To develop your applications with JPA, you can use any relational database.

Domain Model Classes

Your domain model should consist of classes representing entities–lightweight persistent domain objects. The easiest way to define an entity class is by using the `@Entity` annotation (see Using Metadata Annotations), as the following example shows:

```
@Entity
```

```
public class Employee implements Serializable {
...
}
```

For more information on entities, see the following:

- Section 2.1 "Requirements on the Entity Class" of the JPA Specification
- Section 8.1 "Entity" of the JPA Specification
- Considering JPA Entity Architecture
- Configuring an Entity

persistence.xml File

Use the `persistence.xml` file to package your entities.

For more information and examples, see the following:

- Section 6.2.1 "persistence.xml file" of the JPA Specification
- What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties

Object Relational Mapping Metadata

Object relational mapping metadata specifies the mapping of your domain model classes to the database.

You can express this metadata in the form of annotations and/or XML.

Metadata Annotations and ORM.xml File

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. Annotations for object relational mapping are in the `javax.persistence` package. For more information and examples, see Chapter 8 "Metadata Annotations" of the JPA Specification

An object relational mapping XML file is optional. If you choose to provide one, then it should contain mapping information for the classes listed in it. The persistence provider loads an `orm.xml` file (or other mapping file) as a resource. If you provide a mapping file, the classes and mapping information specified in the mapping file will be used. For more information and examples, see the following:

- Using XML
- Section 6.2.1.6 "mapping-file, jar-file, class, exclude-unlisted-classes" of the JPA Specification

Overriding Annotations with XML

XML mapping metadata may combine with and override annotation metadata. For more information and examples, see the following:

- Section 10.1 "XML Overriding Rules" of the JPA Specification

- Overriding Annotations with XML

Advantages and Disadvantages of Using Annotations

Metadata annotations are relatively simple to use and understand. They provide in-line metadata located with the code that this metadata is describing–you do not need to replicate the source code context of where the metadata applies.

On the other hand, annotations unnecessarily couple the metadata to the code. Thus, changes to metadata require changing the source code.

Advantages and Disadvantages of Using XML

The following are advantages of using XML:

- 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 are the complexity and the need for replication of the code context.

Persistence Provider

The persistence provider supplies the implementation of the JPA specification.

The persistence provider handles the object relational mapping of the relationships, including their loading and storing to the database (as specified in the metadata of the entity class), and the referential integrity of the relationships (as specified in the database).

For example, the EclipseLink persistence provider ensures that relational descriptors are created for annotated objects, as well as mappings are created based on annotations.

Persistence Application Code

To manage entities (see Domain Model Classes) in your persistence application, you need to obtain an entity manager from an `EntityManagerFactory`. How you get the entity manager and its factory largely depends on the Java environment in which you are developing your application.

Container-Managed Entity Manager

In the Java EE environment, you acquire an entity manager by injecting it using the `@PersistenceContext` annotation (dependency injection), as the Obtaining an Entity Manager Through Dependency Injection example shows, or using a direct lookup of the entity manager in the JNDI namespace, as the Performing JNDI Lookup of an Entity Manager example shows.

Obtaining an Entity Manager Through Dependency Injection

```
@PersistenceContext
 public EntityManager em;
```

Note: You can only use the `@PersistenceContext` annotation injection on session beans, servlets and JSP.

Performing JNDI Lookup of an Entity Manager

```
@Stateless
 @PersistenceContext(name="ProjectEM", unitName="Project")
 public class ProjectSessionBean implements Project {
      @Resource
      SessionContext ctx;

      public void makeCurrent() {
           EntityManager em = (EntityManager)ctx.lookup("ProjectEM");
           ...
      }
 }
```

The container would manage the life cycle of this entity manager–your application does not have to create it or close it.

For more information and examples, see the following sections of the JPA Specification:

- Section 3.1 "EntityManager"
- Section 5.2.1 "Obtaining an Entity Manager in the Java EE Environment"
- Section 5.3.1 "Obtaining an Entity Manager Factory in a Java EE Container"

Application-Managed Entity Manager

In the Java SE environment, not the container but the application manages the life cycle of an entity manager. You would create this entity manager using the `EntityManagerFactory`'s method `createEntityManager`. You have to use the `javax.persistence.Persistence` class to bootstrap an `EntityManagerFactory` instance, as this example shows:

Application-Managed Entity Manager in the Java SE Environment

```
public class Employee {
```

```
      public static void main(String[] args) {
            EntityManagerFactory emf =
                  Persistence.createEntityManagerFactory("EmpService");
            EntityManager em = emf.createEntityManager();
      ...
            em.close();
            emf.close();
 }
```

Notice that you need to explicitly close the entity manager and the factory.

In the Java EE environment, you can use the application-managed entity managers as well. You would create it using the `@PersistenceUnit` annotation to declare a reference to the `EntityManagerFactory` for a persistence unit, as the following example shows:

```
@PersistenceUnit
 EntityManagerFactory emf;
```

Note: You can only use the `@PersistenceContext` annotation injection on session beans, servlets and JSP.

For more information and examples, see the following sections of the JPA Specification:

- Section 5.2.2 "Obtaining an Application-managed Entity Manager"
- Section 5.3.2 "Obtaining an Entity Manager Factory in a Java SE Environment"

Transaction Management

Transactions define when new, changed or removed entities are synchronized to the database.

JPA supports the following two types of transaction management:

- JTA Transaction Management
- Resource-Local Transactions

Container-managed entity managers always use JTA transactions. Application-managed entity managers may use JTA or resource-local transactions. The default transaction type for Java EE application is JTA.

You define the transaction type for a persistence unit and configure it using the `persistence.xml` file (see persistence.xml File).

For more information, see Section 5.5 "Controlling Transactions" of the JPA Specification.

JTA Transaction Management

JTA transactions are the transactions of the Java EE server.

As section 5.5.1 "JTA Entity Managers" of the JPA Specification defines, "An entity manager whose transactions are controlled through JTA is a JTA entity manager. A JTA entity manager participates in the current JTA transaction, which is begun and committed external to the entity manager and propagated to the underlying resource manager."

Resource-Local Transactions

Resource-local transactions are the native transactions of the JDBC drivers that are referenced by a persistence unit. Your application explicitly controls these transactions. Your application interacts with the resource-local transactions by acquiring an implementation of the `EntityTransaction` interface from the entity manager.

For more information and examples, see the following sections of the JPA Specification.

- Section 5.5.2 "Resource-local Entity Managers"
- Section 5.5.2.1 "The EntityTransaction Interface"

Related Topics

This section introduces EclipseLink implementation of Java Persistence API.

As a specification, JPA needs to be implemented by vendors or open source projects.

EclipseLink provides a complete, EJB 3.0-compliant JPA implementation. It provides complete compliance for all of the mandatory features, many of the optional features, and some additional features. The additional nonmandatory functionality includes the following:

- 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.

EclipseLink offers support for deployment within an EJB 3.0 container. This includes Web containers and other non-EJB 3.0 Java EE containers. For more information, see Deploying an EclipseLink JPA Application.

Through its pluggable persistence capabilities EclipseLink can function as the persistence provider in a compliant EJB 3.0 container.

For more information, see Introduction to EclipseLink.

You can perform object-relational mapping with EclipseLink JPA by the means of doing the following:

- Using Metadata Annotations
- Using XML
- Overriding and Merging XML
- Defaulting Properties
- Configuring an Entity
- Declaring Basic Property Mappings
- Mapping Relationships
- Mapping Inheritance
- Using Embedded Objects

Using Metadata 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.

You can use annotations to configure the persistent behavior of your entities. For example, to designate a Java class as a JPA entity, use the `@Entity` annotation (see Section 8.1 "Entity" of the JPA Specification) as follows:

```
@Entity
 public class Employee implements Serializable {
       ...
 }
```

You can apply annotations at three different levels: at the class, method, and field levels.

For more information and examples, see the following:

- Chapter 8 "Metadata annotations" of the JPA Specification
- Using EclipseLink JPA Extensions

EclipseLink defines a set of proprietary annotations. You can find them in the `org.eclipselink.annotations` package.

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

Using XML

You can use XML mapping metadata on its own, or in combination with annotation metadata, or you can use it to override the annotation metadata.

If you choose to include one or more mapping XML files in your persistence unit, each file must conform and be valid against the `orm_1_0.xsd` schema located at `http://java.sun.com/xml/ns/persistence/orm_1_0.xsd`. This schema defines a namespace called `http://java.sun.com/xml/ns/persistence/orm` that includes all of the ORM elements that you can use in your mapping file.

This example shows a typical XML header for a mapping file:

XML Header for Mapping File

```
<?xml version="1.0" encoding="UTF-8"?>
 <entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xdi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                                 http://java.sun.com/xml/ns/persistence/orm_1_
0.xsd"
      version="1.0">
```

The root element of the mapping file is called `entity-mappings`. All object relational XML metadata is contained within this element. The subelements of `entity-mappings` can be categorized into four main scoping and functional groups: persistence unit defaults, mapping file defaults, queries and generators, and managed classes and mappings. There is also a special setting that determines whether annotations should be considered in the metadata for the persistence unit.

For more information and examples, see Section 10.1 "XML Overriding Rules" of the JPA Specification.

EclipseLink provides a set of persistence unit properties that you can specify in your `persistence.xml` file, or in a property map file (`eclipse.persistence.config.PersistenceUnitProperties`). For more information, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

Similar to EclipseLink annotation extensions, EclipseLink persistence unit properties expose some features of EclipseLink that are currently not available through the use of JPA metadata.

For more information, see Using EclipseLink JPA Extensions.

Overriding and Merging XML

You can use EclipseLink's native metadata xml file, EclipseLink-ORM.XML, to override mappings defined in JPA's configuration file `orm.xml` and provide EclipseLink with extended ORM features. For more information on JPA extensions for mapping, see Using EclipseLink JPA Extensions.

The EclipseLink-ORM.XML file defines the object-relational mapping metadata for EclipseLink. It is built from the existing `orm.xml` file which makes it more intuitive, requires minimum configuration, and easy to override. For more information, see Section 10.1 "XML Overriding Rules" of the JPA Specification.

To override `orm.xml` file's mapping, you must define the `META-INF/eclipselink-orm.xml` file in the project. When both `orm.xml` and `eclipselink-orm.xml` are specified, 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 they are no conflicting entities.

Note: The order of files defined in `persistence.xml` does not define the order of their processing. The files are processed, merged and overidden as required. For more information, see Overriding and Merging Examples.

The EclipseLink-ORM.XML file can be referenced for inclusion in a persistence unit's metadata through any of the following files or methods:

File/Method	Description
`META-INF/eclipselink-orm.xml`	Provides mapping overriding capabilities.
`META-INF/orm.xml`	The default ORM file provided with JPA.
Referenced as persistence unit mapping file in `persistence.xml`	Does not provide mapping overriding capability, but can be used for merging mapping files.

Overriding and Merging Examples

Example 1

- `META-INF/orm.xml` - defines Entity A with the mappings b and c.
- `META-INF/eclipselink-orm.xml` - defines Entity A with the mappings for c and d.
- Result - Entity A will contain the mapping b (from `orm.xml`), mapping c and d (from `eclipselink-orm.xml`)

Example 2

- `META-INF/orm.xml` - defines Entity A with the mappings b and c
- `META-INF/some-other-mapping-file.xml` - defines Entity B with mappings a and b
- `META-INF/eclipselink-orm.xml` - defines Entity A with the mappings for c and d and Entity B with the mapping b and c
- Result
  - Entity A will contain the mapping b (from `orm.xml`), mapping c and d (from `eclipselink-orm.xml`)
  - Entity B will contain the mapping a (from `some-other-mapping-file`), mappings b and c (from `eclipselink-orm.xml`)

Example 3

- `META-INF/orm.xml` - defines Entity A with the mappings b and c.
- `META-INF/eclipse-orm.xml` - defines Entity A with the mappings c and d.
- `META-INF/some-other-mapping-file.xml` - defines Entity A with the mapping x.
- Result - Entity A will contain the mapping b (from `orm.xml`), mapping c and d (from `eclipselink-orm.xml`) and mapping x (from some-other-mapping-file)

Example 4

- `META-INF/orm.xml` - defines Entity A with the mappings b and c.
- `META-INF/extensions/eclipselink-orm.xml` - defines defines Entity A with the mappings c and d.
  Note: This file is added through a <mapping-file> tag in the `persistence.xml`
- Result - Exception generated for conficting specifications for mapping c.

Example 5

- `META-INF/orm.xml` - defines Entity A with the mappings b and c.
- `META-INF/jpa-mapping-file.xml` - defines Entity A with the mappings a and d.
- `META-INF/extensions/eclipse-mapping-file.xml` - defines defines Entity A with the mappings c and d.
- Result - Exception generated for conficting specifications for mapping c or d (which ever is processed first).

Overriding and Merging Rules

The following sections outlines elements defined in `orm.xml` and their specific overriding in greater detail.

Persistence Unit Metadata

In EclipseLink-ORM.XML, a persistence-unit-metadata specification merges or overrides the values of existing persistence-unit-metadata specification.

entity-mappings/persistence-unit-metadata	Rule	Description
xml-mapping-metadata-complete	Full override	If specified, the complete set of mapping metadata for the persistence unit is contained in the XML mapping files for the persistence unit.
persistence-unit-defaults/schema	Full override	If a schema setting exists, then the EclipseLink-ORM.XML's schema setting overrides the existing setting, or creates a new schema setting.
persistence-unit-defaults/catalog	Full override	If a catalog setting exists, then the EclipseLink-ORM.XML's catalog setting overrides the existing setting, or creates a new catalog setting
persistence-unit-defaults/access	Full override	If an access setting exists, then the EclipseLink-ORM.XML's access setting overrides the existing setting, or creates a new access setting.
entity-mappings/persistence-unit-metadata/persistence-unit-defaults/cascade-persist	Full override	If a cascade-persist setting exists, then the EclipseLink-ORM.XML's cascade-persist setting overrides the existing setting, or creates a new cascade-persist setting.
entity-mappings/persistence-unit-metadata/persistence-unit-defaults/entity-listeners	Merge	If an entity-listeners exists, then the EclipseLink-ORM.XML's entity-listeners will be merged with the list of all entity-listeners from the persistence unit.

Entity Mappings

Entities, embeddables and mapped superclasses are defined within entity-mappings. EclipseLink-ORM.XML's entities, embeddables and mapped superclasses are added to the persistence unit. The following table describes the top-level elements of the entity-mappings sections:

entity-mappings/	Rule	Description
package	None	The package element specifies the package of the classes listed within the subelements and attributes of the same mapping file only. It is only applicable to those entities that are fully defined within the EclipseLink-ORM.XML file, else its usage remains local and is same as described in the JPA specification.
catalog	None	The catalog element applies only to the subelements and attributes listed within the EclipseLink-ORM.XML file that are not an extension to another mapping file. Otherwise, the use of the catalog element within the EclipseLink-ORM.XML file remains local and is same as described in the JPA specification.
schema	None	The schema element applies only to the subelements and attributes listed within the EclipseLink-ORM.XML file that are not an extension to another mapping file. Otherwise, the use of the schema element within the EclipseLink-ORM.XML file remains local and is same as described in the JPA specification.
access	None	The access element applies only to the subelements and attributes listed within the EclipseLink-ORM.XML file that are not an extension to another mapping file. Otherwise, the use of the access element within the EclipseLink-ORM.XML file remains local and is same as described in the JPA specification.
sequence-generator	Full override	A sequence-generator is unique by name. The sequence-generator defined in the EclipseLink-ORM.XML will override a sequence-generator of the same name defined in another mapping file. Outside of the overriding case, an exception is thrown if two or more sequence-generators with the same name are defined in one or across multiple mapping files.
table-generator	Full override	A table-generator is unique by name. The table-generator defined in the EclipseLink-ORM.XML will override a table-generator of the same name defined in another mapping file. Outside of the overriding case, an exception is thrown if two or more table-generators with the same name are defined in one or across multiple mapping files.
named-query	Full override	A named-query is unique by name. The named-query defined in the EclipseLink-ORM.XML will override a named-query of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more named-querys with the same

		name are defined in one or across multiple mapping file.
named-native-query	Full override	A named-native-query is unique by name. The named-native-query defined in the EclipseLink-ORM.XML will override a named-native-query of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more named-native-querys with the same name are defined in one or across multiple mapping files.
sql-result-set-mapping	Full override	A sql-result-set-mapping is unique by name. The sql-result-set-mapping defined in the EclipseLink-ORM.XML will override a sql-result-set-mapping of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more sql-result-set-mapping entities with the same name are defined in one or across multiple mapping files.

Mapped Superclass

A mapped-superclass can be defined completely, or with specific elements to provide extensions to a mapped-superclass from another mapping file. The following table lists individual override and merging rules:

entity-mappings/mapped-superclass	Rule	Description
id-class	Full override	If an id-class setting exists, then the EclipseLink-ORM.XML's id-class setting overrides the existing setting, or creates a new id-class setting.
exclude-default-listeners	Full override	If an exclude-default-listeners setting exists, then the EclipseLink-ORM.XML's exclude-default-listeners setting will be applied. If the exclude-default-listeners setting is not specified, it will not override an existing setting, that is essentially turning it off.
exclude-superclass-listeners	Full override	If an exclude-superclass-listeners setting exists, then the EclipseLink-ORM.XML's exclude-superclass-listeners setting will be applied. If exclude-superclass-listeners setting is not specified, it will not override an existing setting, that is essentially turning it off.

entity-listeners	Merge and full override	If an entity-listeners setting exists, then the EclipseLink-ORM.XML's entity-listeners setting will override and merge with an existing setting, or creates a new entity-listeners setting all together. **Note:** An entity listener override must be complete. All lifecycle methods of that listener must be specified and no merging of individual lifecycle methods of an entity listener is allowed. The class name of the listener is the key to identify the override.
pre-persist	Full override	If a pre-persist setting exists, then the EclipseLink-ORM.XML's pre-persist setting overrides the existing setting, or creates a new pre-persist setting.
post-persist	Full override	If a post-persist setting exists, then the EclipseLink-ORM.XML's post-persist setting overrides the existing setting, or creates a new post-persist setting.
pre-remove	Full override	If a pre-remove setting exists, then the EclipseLink-ORM.XML's pre-remove setting overrides the existing setting, or creates a new pre-remove setting.
post-remove	Full override	If a post-remove setting exists, then the EclipseLink-ORM.XML's post-remove setting overrides the existing setting, or creates a new post-remove setting.
pre-update	Full override	If a pre-update setting exists, then the EclipseLink-ORM.XML's pre-update setting overrides the existing setting, or creates a new pre-update setting.
post-update	Full override	If a post-update setting exists, then the EclipseLink-ORM.XML's post-update setting overrides the existing setting, or creates a new post-update setting.
post-load	Full override	If a post-load setting exists, then the EclipseLink-ORM.XML's post-load setting overrides the existing setting, or creates a new post-load setting.
attributes	Merge and mapping level override	If the attribute settings (id, embedded-id, basic, version, many-to-one, one-to-many, one-to-one, many-to-many, embedded, transient) exist at the mapping level, then the EclipseLink-ORM.XML's attributes merges or overrides the existing settings, else creates new attributes.
class	None	
access	Full override	If an access setting exists, then the EclipseLink-ORM.XML's access setting overrides the existing setting, or creates a new access setting. It also overrides the default class

		setting.
metadata-complete	Full override	If a metadata-complete setting exists, then the EclipseLink-ORM.XML's metadata-complete setting will be applied. If metadata-complete setting is not specified, it will not override an existing setting, that is essentially turning it off.

Entity override and merging rules

An entity can be defined completely, or with specific elements to provide extensions to an entity from another mapping file. The following table lists individual override and merging rules:

entity-mappings/entity	Rule	Comments
table	Full override	The table definition overrides any other table setting (with the same name) for this entity. There is no merging of individual table values.
secondary-table	Full override	The secondary-table definition overrides another secondary-table setting (with the same name) for this entity. There is no merging of individual secondary-table(s) values.
primary-key-join-column	Full override	The primary-key-join-column(s) definition overrides any other primary-key-join-column(s) setting for this entity. There is no merging of the primary-key-join-column(s). The specification is assumed to be complete and these primary-key-join-columns are the source of truth.
id-class	Full override	If an id-class setting exists, then the EclipseLink-ORM.XML's id-class setting overrides the existing setting, or creates a new id-class setting.
inheritance	Full override	If an inheritance setting exists, then the EclipseLink-ORM.XML's inheritance setting overrides the existing setting, or creates a new inheritance setting.
discriminator-value	Full override	If a discriminator-value setting exists, then the EclipseLink-ORM.XML's discriminator-value setting overrides the existing setting, or creates a new discriminator-value setting.
discriminator-column	Full override	If a discriminator-column setting exists, then the EclipseLink-ORM.XML's discriminator-column setting overrides the existing setting, or creates a new discriminator-column setting.

sequence-generator	Full override	A sequence-generator is unique by name. The sequence-generator defined in EclipseLink-ORM.XML overrides sequence-generator of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more sequence-generators with the same name are defined in one or across multiple mapping files.
table-generator	Full override	A table-generator is unique by name. The table-generator defined in EclipseLink-ORM.XML overrides table-generator of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more table-generators with the same name are defined in one or across multiple mapping files.
named-query	Merge and full override	A named-query is unique by name. The named-query defined in EclipseLink-ORM.XML overrides named-query of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more named-query elements with the same name are defined in one or across multiple mapping files.
named-native-query	Merge and full override	A named-native-query is unique by name. The named-native-query defined in EclipseLink-ORM.XML overrides named-native-query of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more named-native-query elements with the same name are defined in one or across multiple mapping files.
sql-result-set-mapping	Merge and full override	A sql-result-set-mapping is unique by name. The sql-result-set-mapping defined in EclipseLink-ORM.XML overrides sql-result-set-mapping of the same name defined in other mapping files. Outside of the overriding case, an exception is thrown if two or more sql-result-set-mapping elements with the same name are defined in one or across multiple mapping files.
exclude-default-listeners	Full override	If an exclude-default-listeners setting exists, then the EclipseLink-ORM.XML's exclude-default-listeners setting will be applied. If an exclude-default-listeners setting is not specified, it will not override an existing setting, that is essentially turning it off.
exclude-superclass-listeners	Full override	If an exclude-superclass-listeners setting exists, then the EclipseLink-ORM.XML's exclude-superclass-listeners setting will be applied. If an exclude-superclass-listeners setting is not specified, it will not override an existing setting, that is essentially turning it off.

entity-listeners	Full override	If an entity-listeners setting exists, then the EclipseLink-ORM.XML's entity-listeners setting will override and merge with an existing setting, or creates a new entity-listeners setting all together. **Note:** An entity listener override must be complete. All lifecycle methods of that listener must be specified and no merging of individual lifecycle methods of an entity listener is allowed. The class name of the listener is the key to identify the override.
pre-persist	Full override	If a pre-persist setting exists, then the EclipseLink-ORM.XML's pre-persist setting overrides the existing setting, or creates a new pre-persist setting.
post-persist	Full override	If a post-persist setting exists, then the EclipseLink-ORM.XML's post-persist setting overrides the existing setting, or creates a new post-persist setting.
pre-remove	Full override	If a pre-remove setting exists, then the EclipseLink-ORM.XML's pre-remove setting overrides the existing setting, or creates a new pre-remove setting.
post-remove	Full override	If a post-remove setting exists, then the EclipseLink-ORM.XML's post-remove setting overrides the existing setting, or creates a new post-remove setting.
pre-update	Full override	If a pre-update setting exists, then the EclipseLink-ORM.XML's pre-update setting overrides the existing setting, or creates a new pre-update setting.
post-update	Full override	If a post-update setting exists, then the EclipseLink-ORM.XML's post-update setting overrides the existing setting, or creates a new post-update setting.
post-load	Full override	If a post-load setting exists, then the EclipseLink-ORM.XML's post-load setting overrides the existing setting, or creates a new post-load setting.
attributes	Merge and mapping level override	If the attribute settings (id, embedded-id, basic, version, many-to-one, one-to-many, one-to-one, many-to-many, embedded, transient) exist at the mapping level, then the EclipseLink-ORM.XML's attributes merges or overrides the existing settings, else creates new attributes.
association-override	Merge and full override	If an association-override setting exists, then the EclipseLink-ORM.XML's association-override setting overrides the existing setting, or creates a new association-override setting.
name	Full override	If a name setting exists, then the EclipseLink-ORM.XML's name setting overrides the existing setting, or creates a new

		name setting.
class	None	
access	Full override	If an access setting exists, then the EclipseLink-ORM.XML's access setting overrides the existing setting, or creates a new access setting. It also overrides the default class setting.
metadata-complete	Full override	If a metadata-complete setting exists, then the EclipseLink-ORM.XML's metadata-complete setting will be applied. If a metadata-complete setting is not specified, it will not override an existing setting, that is essentially turning it off.

Embeddable

An embeddable can be defined wholely or may be defined so as to provide extensions to an embeddeable from another mapping file. Therefore, we will allow the merging of that classes metadata. The individual override rules for that metadata is tabled below.

entity-mappings/embeddable	Rule	Description
attributes	Override and merge	If the attribute settings (id, embedded-id, basic, version, many-to-one, one-to-many, one-to-one, many-to-many, embedded, transient) exist at the mapping level, then the EclipseLink-ORM.XML's attributes merges or overrides the existing settings, or creates new attributes.
class	None	
access	Full override	If an access setting exists, then the EclipseLink-ORM.XML's access setting overrides the existing setting, or creates a new access setting. It also overrides the default class setting.
metadata-complete	Full override	If a metadata-complete setting exists, then the EclipseLink-ORM.XML's metadata-complete setting will be applied. If a metadata-complete setting is not specified, it will not override an existing setting, that is essentially turning it off.

Defaulting Properties

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*.

Note: You should be familiar with the defaults to be able to change the

behavior when necessary.

Configuring an Entity

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

Configuring an Entity 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, 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

You can also use these annotations to fine-tune how your database maintains the identity of your entities.

For more information on the EclipseLink artifacts configured by these JPA metadata, refer to Configuring Primary Keys.

@Id

Use the `@Id` annotation to designate one or more persistent fields or properties as the entity's primary key.

For each entity, you must designate at least one of the following:

- one `@Id`
- one `@EmbeddedId`
- multiple `@Id` and an `@IdClass`

Note: The last option in the preceding list – `@Id` and `@IdClass` combination – is applicable to composite primary key configuration.

The `@Id` annotation does not have attributes.

By default, EclipseLink persistence provider chooses the most appropriate primary key generator (see @GeneratedValue) and is responsible for managing primary key values: you do not need to take any further action if you are satisfied with the persistence provider's default key generation mechanism.

This example shows how to use this annotation to designate the persistent field `empID` as the primary key of the `Employee` table.

Usage of @Id Annotation

```
@Entity
 public class Employee implements Serializable {
      @Id
      private int empID;
      ...
 }
```

The `@Id` annotation supports the use of EclipseLink converters (see Using EclipseLink JPA Converters).

For more information and examples, see Section 9.1.8 "Id Annotation" of the JPA Specification.

@IdClass

Use the `@IdClass` annotation to specify a composite primary key class (usually made up of two or more primitive, JDK object types or Entity types) for an entity or MappedSuperclass.

Note: Composite primary keys typically arise during mapping from legacy databases when the database key is comprised of several columns.

A composite primary key class has the following characteristics:

- It is a POJO class.
- It is a public class with a public no-argument constructor.
- If you use property-based access, the properties of the primary key class are public or protected.
- It is serializable.
- It defines `equals` and `hashCode` methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.
- Its fields or properties must correspond in type and name to the entity primary key fields or properties annotated with @Id.

Alternatively, you can make the composite primary key class an embedded class owned by the entity (see @EmbeddedId).

The `@IdClass` annotation has a required attribute `value` that you set to the class to specify this class as a composite primary key class (see @AttributeOverride).

The Nonembedded Composite Primary Key Class example shows a nonembedded composite primary key class. In this class, fields `empName` and `birthDay` must correspond in name and type to properties

in the entity class. The Usage of @IdClass Annotation example shows how to configure an entity with this nonembedded composite primary key class using the `@IdClass` annotation. Because entity class fields `empName` and `birthDay` are used in the primary key, you must also annotate them using the `@Id` annotation (see @Id).

Nonembedded Composite Primary Key Class

```
public class EmployeePK implements Serializable {

      private String empName;
      private Date birthDay;

      public EmployeePK() {
      }

      public String getName() {
            return this.empName;
      }

      public void setName(String name) {
            this.empName = name;
      }

      public long getDateOfBirth() {
            return this.birthDay;
      }

      public void setDateOfBirth(Date date) {
            this.birthDay = date;
      }

      public int hashCode() {
            return (int)this.empName.hashCode();
      }

      public boolean equals(Object obj) {
            if (obj == this) return true;
            if (!(obj instanceof EmployeePK)) return false;
            if (obj == null) return false;
            EmployeePK pk = (EmployeePK) obj;
            return pk.birthDay = this.birthDay &&
pk.empName.equals(this.empName);
      }
 }
```

Usage of @IdClass Annotation

```
@IdClass (EmployeePK.class)
 @Entity
 public class Employee implements Serializable{
```

```
    @Id String empName;
    @Id Date birthDay;
    ...
}
```

For more information and examples, see Section 9.1.15 "IdClass Annotation" of the JPA Specification.

@EmbeddedId

Use the `@EmbeddedId` annotation to specify an embeddable composite primary key class (usually made up of two or more primitive or JDK object types) owned by the entity.

Note: Composite primary keys typically arise during mapping from legacy databases when the database key is comprised of several columns.

A composite primary key class has the following characteristics:

- It is a POJO class.
- It is a public class with a public no-argument constructor.
- If you use property-based access, the properties of the primary key class are public or protected.
- It is serializable.
- It defines `equals` and `hashCode` methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.

Alternatively, you can make the composite primary key class a nonembedded class (see @IdClass).

The `@EmbeddedId` annotation does not have attributes.

This example shows a typical composite primary key class annotated with `@Embeddable`. The Usage of @EmbeddedId Annotation example shows how to configure an entity with this embeddable composite primary key class using the `@EmbeddedId` annotation.

Embeddable Composite Primary Key Class

```
public class EmployeePK implements Serializable {

    private String empName;
    private long empID;

    public EmployeePK() {
    }

    public String getName() {
        return this.empName;
    }
```

```
      public void setName(String name) {
            this.empName = name;
      }

      public long getId() {
            return this.empID;
      }

      public void setId(long id) {
            this.empID = id;
      }

      public int hashCode() {
            return (int)this.empName.hashCode()+ this.empID;
      }

      public boolean equals(Object obj) {
            if (obj == this) return true;
            if (!(obj instanceof EmployeePK)) return false;
            if (obj == null) return false;
            EmployeePK pk = (EmployeePK) obj;
            return pk.empID == this.empID && pk.empName.equals(this.empName);
      }
 }
```

Usage of @EmbeddedId Annotation

```
@Entity
 public class Employee implements Serializable{

      EmployeePK primaryKey;

      public Employee {
      }

      @EmbeddedId
      public EmployeePK getPrimaryKey() {
            return primaryKey:
      }

      public void setPrimaryKey(EmployeePK pk) {
            primaryKey = pk;
      }

      ...
 }
```

For more information and examples, see Section 9.1.14 "EmbeddedId Annotation" of the JPA Specification.

@GeneratedValue

Use the `@GeneratedValue` annotation to enable EclipseLink persistence provider to generate unique identifiers for entity primary keys (see @Id).

This annotation lets you do the following:

- override the type of identity value generation selected by the persistence provider for your database if you feel another generator type is more appropriate for your database or application;
- override the primary key generator name selected by the persistence provider if this name is awkward, a reserved word, incompatible with a preexisting data model, or invalid as a primary key generator name in your database.

The `@GeneratedValue` annotation has the following attributes:

- `generator` – The default value of this attribute is the name that EclipseLink persistence provider assigns to the primary key generator it selects.
  If the generator name is awkward, a reserved word, incompatible with a preexisting data model, or invalid as a primary key generator name in your database, set the value of this attribute to the `String` generator name you want to use.

  You are not required to specify the value of this attribute.
- `strategy` – By default, EclipseLink persistence provider chooses the type of primary key generator that is most appropriate for the underlying database.
  If you feel that another generator type is more appropriate for your database or application, set the value of this attribute to one of the following enumerated values of the `GenerationType` enumerated type:
  - `AUTO` (default) – specify that EclipseLink persistence provider should choose a primary key generator that is most appropriate for the underlying database.

    Note: By default, EclipseLink chooses the `TABLE` strategy using a table named `SEQ_GEN_TABLE`, with `SEQ_NAME` and `SEQ_COUNT` columns, with `allocationSize` of 50 and `pkColumnValue` of `SEQ_GEN`. The default `SEQUENCE` used is database sequence `SEQ_GEN_SEQUENCE` with `allocationSize` of 50. Note that the database sequence increment must match the allocation size.

  - `TABLE` – specify that EclipseLink persistence provider assign primary keys for the entity using an underlying database table to ensure uniqueness (see @TableGenerator).
  - `SEQUENCE` – specify that EclipseLink persistence provider use a database sequence (see @SequenceGenerator).

    Note: `SEQUENCE` strategy is only supported on Oracle Database.

  - `IDENTITY` – specify that EclipseLink persistence provider use a database identity column. Setting this value will indicate to the persistence provider that it must reread the inserted row from the table after an insert has occurred. This will allow it to obtain the newly generated identifier from the database and put it into the in-memory entity that was just persisted. The identity must be defined as part of the database schema for the primary key column. Identity generation may not be shared across multiple entity types.

    Note: `IDENTITY` strategy is supported on Sybase, DB2, SQL Server,

MySQL, Derby, JavaDB, Informix, and Postgres databases.

Note: There is a difference between using `IDENTITY` and other id generation strategies: the identifier will not be accessible until after the insert has occurred – it is the action of inserting that caused the identifier generation. Due to the fact that insertion of entities is most often deferred until the commit time, the identifier would not be available until after the transaction has been committed.

Note: We do not recommend using the `IDENTITY` strategy for it does not support preallocation.

You are not required to specify the value of the `strategy` attribute. This example shows how to use automatic id generation. This will cause EclipseLink persistence provider to create an identifier value and insert it into the `id` field of each `Employee` entity that gets persisted.

Using Automatic Id Generation

```
@Entity
 public class Employee implements Serializable {

       @Id
       @GeneratedValue(strategy=GenerationType.AUTO)
       private int id;
       ...
 }
```

Caution: Be careful when using the automatic id generation: the persistence provider has to pick its own strategy to store the identifiers, but it needs to have a persistent resource, such as a table or a sequence, to do so. The persistence provider cannot always rely upon the database connection that it obtains from the server to have permissions to create a table in the database. This is usually a privileged operation that is often restricted to the DBA. There will need to be a creation phase or schema generation to cause the resource to be created before the `AUTO` strategy can function.

For more information and examples, see Section 9.1.9 "GeneratedValue Annotation" of the JPA Specification.

Configuring 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.

For more information, see Table Sequencing.

@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.

The `@SequenceGenerator` annotation has the following attributes:

- `name` – The name of the generator must match the name of a `GeneratedValue` with its `strategy` attribute set to `SEQUENCE`.

  You are required to specify the value of this attribute.
- `allocationSize` – By default, EclipseLink persistence provider uses an allocation size of 50.

  The value of this attribute must match the increment size on the database sequence object. If this allocation size does not match your application requirements or database performance parameters, set this attribute to the `int` value you want.

  You are not required to specify the value of the `allocationSize` attribute.
- `initialValue` – By default, EclipseLink persistence provider starts all primary key values from 0.
  If this does not match an existing data model, set this attribute to the `int` value you want.

  You are not required to specify the value of the `initialValue` attribute.
- `sequenceName` – By default, EclipseLink persistence provider assigns a sequence name of its own creation.

  The `sequenceName` defaults to the name of the `SequenceGenerator`. If you prefer to use an existing or predefined sequence, set `sequenceName` to the `String` name you want.

  You are not required to specify the value of the `sequenceName` attribute.

This example shows how to use this annotation to specify the allocation size for the `SEQUENCE` primary key generator named `Cust_Seq`.

Usage of @SequenceGenerator

```
@Entity
 public class Employee implements Serializable {
      ...
      @Id
      @SequenceGenerator(name="Cust_Seq", allocationSize=25)
      @GeneratorValue(strategy=SEQUENCE, generator="Cust_Seq")
      @Column(name="CUST_ID")
      public Long getId() {
           return id;
      }
      ...
 }
```

For more information and examples, see Section 9.1.37 "SequenceGenerator Annotation" of the JPA Specification.

For more information on the EclipseLink artifacts configured by these JPA metadata, refer to Descriptors and Sequencing.

@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;

The `@TableGenerator` annotation has the following attributes:

- `name` – The name of the generator must match the name of a `GeneratedValue` with its `strategy` attribute set to `TABLE`. The scope of the generator name is global to the persistence unit (across all generator types).

  You are required to specify the value of this attribute.
- `allocationSize` – By default, EclipseLink persistence provider uses an allocation size of 50.
  If this allocation size does not match your application requirements or database performance parameters, set this attribute to the `int` value you want.

  You are not required to specify the value of the `allocationSize` attribute.
- `catalog` – By default, EclipseLink persistence provider uses whatever the default

catalog is for your database.
If the default catalog is inappropriate for your application, set the value of this attribute to the `String` catalog name to use.

You are not required to specify the value of the `catalog` attribute.

- `initialValue` – By default, EclipseLink persistence provider starts all primary key values from 0.
  If this does not match an existing data model, set this attribute to the `int` value you want.

  You are not required to specify the value of the `initialValue` attribute.
- `pkColumnName` – By default, EclipseLink persistence provider supplies a name for the primary key column in the generator table: `"SEQ_NAME"`.
  If this name is inappropriate for your application, set the value of this attribute to the `String` name you want.

  You are not required to specify the value of the `pkColumnName` attribute.
- `pkColumnValue` – By default, EclipseLink persistence provider supplies a suitable primary key value for the primary key column in the generator table: `TableGenereator.name`.
  If this value is inappropriate for your application, set the value of this attribute to the `String` value you want.

  You are not required to specify the value of the `pkColumnValue` attribute.
- `schema` – By default, EclipseLink persistence provider uses whatever the default schema is for your database.
  If this value is inappropriate for your application, set the value of this attribute to the `String` schema name you choose.

  You are not required to specify the value of the `schema` attribute.
- `table` – By default, EclipseLink persistence provider supplies a suitable name for the table that stores the generated id values: `"SEQUENCE"`.
  If this value is inappropriate for your application, set the value of this attribute to the `String` table name you want.

  You are not required to specify the value of the `table` attribute.
- `uniqueConstraints` – By default, EclipseLink persistence provider assumes that none of the columns in the primary key generator table have unique constraints.
  If unique constraints do apply to one or more columns in this table, set the value of this attribute to an array of one or more `UniqueConstraint` instances. For more information, see Section 9.1.4 "UniqueConstraint Annotation" of the JPA Specification.

  You are not required to specify the value of the `uniqueConstraints` attribute.
- `valueColumnName` – By default, EclipseLink persistence provider supplies a suitable name for the column that stores the generated id values: `"SEQ_COUNT"`.
  If the default column name is inappropriate for your application, set the value of this attribute to the `String` column name you want.

  You are not required to specify the value of the `valueColumnName` attribute.

The Usage of @TableGenerator example shows how to use this annotation to specify the allocation size for the `TABLE` primary key generator named `Emp_Gen`.

Usage of @TableGenerator

```
@Entity
 public class Employee implements Serializable {
      ...
      @Id
      @TableGenerator(name="Emp_Gen", allocationSize=1)
      @GeneratorValue(strategy=TABLE, generator="Emp_Gen")
      @Column(name="CUST_ID")
      public Long getId() {
           return id;
      }
      ...
 }
```

Every table that you use for id generation should have two columns – if there are more columns, only two will be used. The first column is of a string type and is used to identify the particular generator sequence. It is the primary key for all of the generators in the table. The name of this column is specified by the `pkColumnName` attribute. The second column is of an integer type and stores the actual id sequence that is being generated. The value stored in this column is the last identifier that was allocated in the sequence. The name of this column is specified by the `valueColumnName` attribute.

Each defined generator represents a row in the table. The name of the generator becomes the value stored in the `pkColumnName` column for that row and is used by EclipseLink persistence provider to look up the generator to obtain its last allocated value.

For more information and examples, see Section 9.1.38 "TableGenerator Annotation" of the JPA Specification.

Configuring Locking

You have the choice between optimistic and pessimistic locking. We recommend using EclipseLink optimistic locking. For more information, see Locking.

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

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 (see Section 9.1.1 "Table Annotation" of the JPA Specification);
- your application does not modify the version property or field.

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`. We recommend using a numeric type.

The `@Version` annotation does not have attributes.

The Usage of @Version Annotation example shows how to use this annotation to specify property `getVersionNum` as the optimistic lock value. In this example, the column name for this property is set to `OPTLOCK` (see Section 9.1.5 "Column Annotation" of the JPA Specification) instead of the default column name for the property.

Usage of @Version Annotation

```
@Entity
 public class Employee implements Serializable {
       ...
       @Version
       @Column(name="OPTLOCK")
       protected int getVersionNum() {
            return versionNum;
       }
       ...
 }
```

The `@Version` annotation supports the use of EclipseLink converters (see Using EclipseLink JPA Converters).

For more information, see the following:

- Section 3.4 "Optimistic Locking and Concurrency" of the JPA Specification
- Section 9.1.17 "Version Annotation" of the JPA Specification
- Using EclipseLink JPA Extensions for Optimistic Locking

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

Declaring Basic Property 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, EclipseLink persistence provider automatically configures a basic mapping for simple types.

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

- @Basic
- @Enumerated
- @Temporal
- @Lob
- @Transient

For more information, see Using EclipseLink JPA Converters.

@Basic

By default, EclipseLink persistence provider automatically configures `@Basic` mapping for most Java primitive types, wrappers of the primitive types, and enumerated types.

EclipseLink uses the default column name format of `<field-name>` or <property-name> in uppercase characters.

You may explicitly place an optional `@Basic` annotation on a field or property to explicitly mark it as persistent.

Note: The `@Basic` annotation is mostly for documentation purposes – it is not required for the field or property to be persistent.

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

- configure the fetch type to `LAZY`;
- configure the mapping to forbid null values (for nonprimitive types) in case null values are inappropriate for your application.

The `@Basic` annotation has the following attributes:

- `fetch` – By default, EclipseLink persistence provider uses a fetch type of `javax.persitence.FetchType.EAGER`: data must be eagerly fetched.If the default is inappropriate for your application or a particular persistent field, set `fetch` to `FetchType.LAZY`: this is a hint to the persistence provider that data should be fetched lazily when it is first accessed (if possible).You are not required to specify the value of this attribute.
  For more information, see What You May Need to Know About EclipseLink JPA Lazy Loading.
- `optional` – By default, EclipseLink persistence provider assumes that the value of all (nonprimitive) fields and properties may be `null`.
  If the default is inappropriate for your application, set this the value of this attribute to `false`.

  You are not required to specify the value of this attribute.

This example shows how to use this annotation to specify a fetch type of `LAZY` for a basic mapping.

Usage of the @Basic Annotation

```
@Entity
 public class Employee implements Serializable {
       ...
       @Basic(fetch=LAZY)
       protected String getName() {
```

```
            return name;
        }
        ...
  }
```

For more information and examples, see Section 9.1.18 "Basic Annotation" of the JPA Specification.

For more information on EclipseLink direct mappings and relationship mappings, see Relational Mapping Types.

@Enumerated

By default, EclipseLink persistence provider persists the ordinal values of enumerated constants.

Use the `@Enumerated` annotation to specify whether EclipseLink persistence provider should persist ordinal or `String` values of enumerated constants if the `String` value suits your application requirements, or to match an existing database schema:

You can use this annotation with the @Basic annotation.

The `@Enumerated` annotation has the following attributes:

- `value` – By default, EclipseLink persistence provider assumes that for a property or field mapped to an enumerated constant, the ordinal value should be persisted. In the Usage of the @Enumerated Annotation example, the ordinal value of `EmployeeStatus` is written to the database when `Employee` is persisted. If you want the `String` value of the enumerated constant persisted, set value to `EnumType.STRING`.

  You are not required to specify the value of this attribute.

Given the enumerated constants in the Enumerated Constants example, the Usage of the @Enumerated Annotation example shows how to use the `@Enumerated` annotation to specify that the `String` value of `SalaryRate` should be written to the database when `Employee` is persisted. By default, the ordinal value of `EmployeeStatus` is written to the database.

Enumerated Constants

```
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}
 public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}
```

Usage of the @Enumerated Annotation

```
@Entity
 public class Employee implements Serializable{
      ...
      public EmployeeStatus getStatus() {
           ...
```

```
        }

        @Enumerated(STRING)
        public SalaryRate getRate() {
              ...
        }
        ...
  }
```

For more information and examples, see Section 9.1.21 "Enumerated Annotation" of the JPA Specification.

@Temporal

Use the `@Temporal` annotation to specify the database type that EclipseLink persistence provider should persist for persistent fields or properties of type `java.util.Date` and `java.util.Calendar` only.

You can use this annotation with the `@Basic` annotation.

The `@Temporal` annotation has the following attributes:

- `value` – Set this attribute to the `TemporalType` that corresponds to database type you want EclipseLink persistence provider to use:
  - `DATE` – equivalent of `java.sql.Date`
  - `TIME` – equivalent of `java.time.Date`
  - `TIMESTAMP` – equivalent of `java.sql.Timestamp`

You are required to specify the value of this attribute.

This example shows how to use this annotation to specify that EclipseLink persistence provider should persist `java.util.Date` field `startDate` as a `DATE` (`java.sql.Date`) database type.

Usage of the @Temporal Annotation

```
@Entity
 public class Employee implements Serializable{
        ...
        @Temporal(DATE)
        protected java.util.Date startDate;
        ...
  }
```

For more information and examples, see Section 9.1.20 "Temporal Annotation" of the JPA Specification.

@Lob

By default, EclipseLink persistence provider assumes that all persistent data can be represented as typical database data types.

Use the `@Lob` annotation with the `@Basic` mapping to specify that a persistent property or field should be persisted as a large object to a database-supported large object type.

A `Lob` may be either a binary or character type. The persistence provider infers the `Lob` type from the type of the persistent field or property.

For `String` and character-based types, the default is `Clob`. In all other cases, the default is `Blob`.

You can also use the `@Column` attribute `columnDefinition` (see Section 9.1.5 "Column Annotation" of the JPA Specification) to further refine the `Lob` type.

The `@Lob` annotation does not have attributes.

This example shows how to use this `@Lob` annotation to specify that persistent field `pic` should be persisted as a `Blob`.

Usage of the @Lob Annotation

```
@Entity
 public class Employee implements Serializable {
       ...
       @Lob
       @Basic(fetch=LAZY)
       @Column(name="EMP_PIC", columnDefinition="BLOB NOT NULL")
       protected byte[] pic;
       ...
 }
```

For more information and examples, see Section 9.1.20 "Temporal Annotation" of the JPA Specification.

@Transient

By default, EclipseLink persistence provider assumes that all the fields of an entity are persistent.

Use the `@Transient` annotation to specify a field or property of an entity that is not persistent (for example, a field or property that is used at run time, but that is not part of the entity's state).

EclipseLink persistence provider will not persist (or create database schema) for a property or field annotated with `@Transient`.

This annotation can be used with `@Entity` (see Section 8.1 "Entity" of the JPA Specification), `@MappedSuperclass`), and `@Embeddable`.

The `@Transient` annotation does not have attributes.

The Usage of the @Transient Annotation example shows how to use the `@Transient` annotation to specify `Employee` field `currentSession` as not persistent. EclipseLink persistence provider will not persist this field.

Usage of the @Transient Annotation

```
@Entity
 public class Employee implements Serializable {
       ...
       @Id
       int id;

       @Transient
       Session currentSession;
       ...
 }
```

For more information and examples, see Section 9.1.16 "Transient Annotation" of the JPA Specification.

Mapping Relationships

EclipseLink persistence provider requires that you map relationships explicitly.

Use the following annotations to specify the type and characteristics of entity relationships to fine-tune how your database implements these relationships:

- @OneToOne
- @ManyToOne
- @OneToMany
- @ManyToMany
- @MapKey
- @MapKey

At end of relationships section should link to the EclipseLink relationships mappings section and state that additional advanced mapping and mapping options are available through EclipseLink's descriptor and mapping API through using a `DescriptorCustomizer`.

For more information, see Relational Mapping Types.

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

@OneToOne

By default, JPA automatically defines a `OneToOne` mapping for a single-valued association to another entity that has one-to-one multiplicity and infers the associated target entity from the type of the object being referenced.

Use the `OneToOne` annotation to do the following:

- configure the fetch type to `LAZY`;
- configure the mapping to forbid null values (for nonprimitive types) in case null values are inappropriate for your application;
- configure the associated target entity, if it cannot be inferred from the type of the object being referenced;
- 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).

The `@OneToOne` annotation has the following attributes:

- `cascade` – By default, JPA does not cascade any persistence operations to the target of the association. Thus, the default value of this attribute is an empty `javax.persitence.CascadeType` array.
  If you want some or all persistence operations cascaded to the target of the association, set the value of this attribute to one or more `CascadeType` instances, including the following:
  - `ALL` – Any persistence operation performed on the owning entity is cascaded to the target of the association.
  - `MERGE` – If the owning entity is merged, the merge is cascaded to the target of the association.
  - `PERSIST` – If the owning entity is persisted, the persist is cascaded target of the association.
  - `REFRESH` – If the owning entity is refreshed, the refresh is cascaded target of the association.
  - `REMOVE` – If the owning entity is removed, the target of the association is also removed.

    You are not required to provide value for this attribute.
- `fetch` – By default, EclipseLink persistence provider uses a fetch type of `javax.persitence.FetchType.EAGER`: this is a requirement on the persistence provider runtime that data must be eagerly fetched.If the default is inappropriate for your application or a particular persistent field, set `fetch` to `FetchType.LAZY`: this is a hint to the persistence provider that data should be fetched lazily when it is first accessed (if possible). We recommend using the FetchType.LAZY on all relationships.
  You are not required to provide value for this attribute.
  For more information, see What You May Need to Know About EclipseLink JPA Lazy Loading.
- `mappedBy` – By default, EclipseLink persistence provider infers the associated target entity from the type of the object being referenced.
  Use the `mappedBy` attribute if the relationship is bidirectional and the target entity has an inverse one-to-one relationship that has already been mapped. You can only use `mappedBy` on the side of the relationship that does not define the foreign key in its table. This is the only way in JPA to define a target foreign key relationship. For example, if the foreign key for the one-to-one is in the target entity's table, you must define the one-to-one mapping on both sides of the relationship and use the `mappedBy` on the target foreign key side. For more

information on target foreign keys, see One-to-One Mapping.

You are not required to specify the value of this attribute.

- `optional` – By default, EclipseLink persistence provider assumes that the value of all (nonprimitive) fields and properties may be `null`.

  The default value of this attribute is `true`.
  If the default is inappropriate for your application, set value of this attribute to `false`.

  You are not required to specify the value of this attribute.
- `targetEntity` – By default, EclipseLink persistence provider infers the associated target entity from the type of the object being referenced.
  If the persistence provider cannot infer the type of the target entity, then set the `targetEntity` element on owning side of the association to the `Class` of the entity that is the target of the relationship.

  You are not required to specify the value of this attribute.

The Usage of @OneToOne Annotation - Customer Class and Usage of @OneToOne Annotation - CustomerRecord Class examples show how to use this annotation to configure a one-to-one mapping between `Customer` (the owning side) and `CustomerRecord` (the owned side).

Usage of @OneToOne Annotation - Customer Class

```
@Entity
 public class Customer implements Serializable {
       ...
       @OneToOne(optional=false)
       @JoinColumn(name="CUSTREC_ID", unique=true, nullable=false,
updatable=false)
       public CustomerRecord getCustomerRecord() {
            return customerRecord;
       }
       ...
 }
```

Note: You have to provide a `@JoinColumn` (see Section 9.1.6 "JoinColumn Annotation" of the JPA Specification) for a `@OneToOne` defining the foreign key. Otherwise, the foreign key will be assumed to be the `<source-field-name>_ <target-primary-key-column>` or `<source-property-name>_ <target-primary-key-column>`.

Use either a `@JoinColumn` or a `@JoinTable` (see Section 9.1.25 "JoinTable Annotation" of the JPA Specification) with the mapping; if you do not specify any of them, EclipseLink will default to `@JoinTable` with the join table name format of `<source-table-name>_<target-table-name>` in uppercase characters, and with columns format of `<source-entity-alias>_<source-primary-key-column>`, `<source-field-name>_ <target-primary-key-column>` (or <source-property-name>_ <target-primary-key-column>) in uppercase characters.

Usage of @OneToOne Annotation - CustomerRecord Class

```
@Entity
 public class CustomerRecord implements Serializable {
      ...
      @OneToOne(optional=false, mappedBy="customerRecord")
      public Customer getCustomer() {
           return customer;
      }
      ...
 }
```

For more information and examples, see Section 9.1.23 "OneToOne Annotation" of the JPA Specification.

For more information on EclipseLink direct mappings and relationship mappings, see Relational Mapping Types.

For more information, see One-to-One Mapping and Configuring a Relational One-to-One Mapping.

@ManyToOne

By default, JPA automatically defines a `ManyToOne` mapping for a single-valued association to another entity class that has many-to-one multiplicity.

Use the `ManyToOne` annotation to do the following:

- configure the fetch type to `LAZY`;
- configure the mapping to forbid null values (for nonprimitive types) in case null values are inappropriate for your application;
- configure the associated target entity, if it cannot be inferred from the type of the object being referenced;
- 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).

The `@ManyToOne` annotation has the following attributes:

- `cascade` – By default, JPA does not cascade any persistence operations to the target of the association. Thus, the default value of this attribute is an empty `javax.persistence.CascadeType` array.
  If you want some or all persistence operations cascaded to the target of the association, set the value of this attribute to one or more `CascadeType` instances, including the following:
  - `ALL` – Any persistence operation performed on the owning entity is cascaded to the target of the association.
  - `MERGE` – If the owning entity is merged, the merge is cascaded to the target of the association.

- `PERSIST` – If the owning entity is persisted, the persist is cascaded target of the association.
- `REFRESH` – If the owning entity is refreshed, the refresh is cascaded target of the association.
- `REMOVE` – If the owning entity is removed, the target of the association is also removed.

  You are not required to provide value for this attribute.

- `fetch` – By default, EclipseLink persistence provider uses a fetch type of javax.persitence.`FetchType.EAGER`: this is a requirement on the persistence provider runtime that data must be eagerly fetched.
  If the default is inappropriate for your application or a particular persistent field, set `fetch` to `FetchType.LAZY`: this is a hint to the persistence provider that data should be fetched lazily when it is first accessed (if possible).

  You are not required to provide value for this attribute.
  For more information, see What You May Need to Know About EclipseLink JPA Lazy Loading.
- `optional` – By default, EclipseLink persistence provider assumes that the value of all (nonprimitive) fields and properties may be `null`.

  The default value of this attribute is `true`.
  If the default is inappropriate for your application, set value of this attribute to `false`.

  You are not required to specify the value of this attribute.
- `targetEntity` – By default, EclipseLink persistence provider infers the associated target entity from the type of the object being referenced.
  If the persistence provider cannot infer the type of the target entity, then set the `targetEntity` element on owning side of the association to the `Class` of the entity that is the target of the relationship.

  You are not required to specify the value of this attribute.

This example shows how to use this annotation to configure a many-to-one mapping between `Customer` (the owned side) and `Order` (the owning side) using generics.

Usage of @ManyToOne Annotation

```
@Entity
 public class Order implements Serializable {
       ...
       @ManyToOne(optional=false)
       @JoinColumn(name="CUST_ID", nullable=false, updatable=false)
       public Customer getCustomer() {
            return customer;
       }
       ...
 }
```

Note: You have to provide a `@JoinColumn` (see Section 9.1.6

"JoinColumn Annotation" of the JPA Specification) for a `@ManyToOne` defining the foreign key. Otherwise, the foreign key will be assumed to be the `<source-field-name>_ <target-primary-key-column>` or <source-property-name>_ <target-primary-key-column>.

Use either a `@JoinColumn` or a `@JoinTable` (see Section 9.1.25 "JoinTable Annotation" of the JPA Specification) with the mapping; if you do not specify any of them, EclipseLink will default to `@JoinTable` with the join table name format of `<source-table-name>_<target-table-name>` in uppercase characters, and with columns format of `<source-entity-alias>_<source-primary-key-column>`, `<source-field-name>_ <target-primary-key-column>` (or <source-property-name>_ <target-primary-key-column>) in uppercase characters.

For more information and examples, see Section 9.1.22 "ManyToOne Annotation" of the JPA Specification.

For more information on EclipseLink direct mappings and relationship mappings, see Relational Mapping Types.

@OneToMany

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 (see Section 9.1.25 "JoinTable Annotation" of the JPA Specification).

The `@OneToMany` annotation has the following attributes:

- `cascade` – By default, JPA does not cascade any persistence operations to the target of the association. Thus, the default value of this attribute is an empty `javax.persitence.CascadeType` array.
  If you want some or all persistence operations cascaded to the target of the association, set the value of this attribute to one or more `CascadeType` instances, including the following:
  - `ALL` – Any persistence operation performed on the owning entity is cascaded to the target of the association.
  - `MERGE` – If the owning entity is merged, the merge is cascaded to the target of the association.
  - `PERSIST` – If the owning entity is persisted, the persist is cascaded target of the association.

- `REFRESH` – If the owning entity is refreshed, the refresh is cascaded target of the association.
- `REMOVE` – If the owning entity is removed, the target of the association is also removed.

  You are not required to provide value for this attribute.

- `fetch` – By default, EclipseLink persistence provider uses a fetch type of javax.persitence.`FetchType.LAZY`: this is a hint to the persistence provider that data should be fetched lazily when it is first accessed (if possible).
  If the default is inappropriate for your application or a particular persistent field, set `fetch` to `FetchType.EAGER`: this is a requirement on the persistence provider runtime that data must be eagerly fetched.

  You are not required to provide value for this attribute.
  For more information, see What You May Need to Know About EclipseLink JPA Lazy Loading.
- `mappedBy` – By default, if the relationship is unidirectional, EclipseLink persistence provider determines the field that owns the relationship.
  If the relationship is bidirectional, then set the `mappedBy` element on the inverse (non-owning) side of the association to the name of the field or property that owns the relationship, as the Usage of @ManyToOne Annotation - Order Class with Generics example shows.

  You are not required to specify the value of this attribute.
- `targetEntity` – By default, if you are using a `Collection` defined using generics, then the persistence provider infers the associated target entity from the type of the object being referenced. Thus, the default is the parameterized type of the `Collection` when defined using generics.
  If your `Collection` does not use generics, then you must specify the entity class that is the target of the association: set the `targetEntity` element on owning side of the association to the `Class` of the entity that is the target of the relationship.

  You are not required to specify the value of this attribute.

The Usage of @OneToMany Annotation - Customer Class with Generics and Usage of @ManyToOne Annotation - Order Class with Generics examples show how to use this annotation to configure a one-to-many mapping between `Customer` (the owned side) and `Order` (the owning side) using generics.

Usage of @OneToMany Annotation - Customer Class with Generics

```
@Entity
 public class Customer implements Serializable {
      ...
      @OneToMany(cascade=ALL, mappedBy="customer")
      public Set<Order> getOrders() {
           return orders;
      }
      ...
 }
```

Usage of @ManyToOne Annotation - Order Class with Generics

```
@Entity
```

```
public class Order implements Serializable {
     ...
     @ManyToOne
     @JoinColumn(name="CUST_ID", nullable=false)
     public Customer getCustomer() {
          return customer;
     }
     ...
}
```

For more information and examples, see Section 9.1.24 "OneToMany Annotation" of the JPA Specification.

For more information on EclipseLink direct mappings and relationship mappings, see Relational Mapping Types.

For more information on EclipseLink one-to-one mappings, see One-to-Many Mapping, and for information on how to configure these mappings, see Configuring a Relational One-to-Many Mapping.

@ManyToMany

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

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

- configure the fetch type 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).

The `@ManyToMany` annotation has the following attributes:

- `cascade` – By default, JPA does not cascade any persistence operations to the target of the association. Thus, the default value of this attribute is an empty `javax.persitence.CascadeType` array.
  If you want some or all persistence operations cascaded to the target of the association, set the value of this attribute to one or more `CascadeType` instances, including the following:
  - `ALL` – Any persistence operation performed on the owning entity is cascaded to the target of the association.
  - `MERGE` – If the owning entity is merged, the merge is cascaded to the target of the association.
  - `PERSIST` – If the owning entity is persisted, the persist is cascaded target of the association.

- `REFRESH` – If the owning entity is refreshed, the refresh is cascaded target of the association.
- `REMOVE` – If the owning entity is removed, the target of the association is also removed.

  You are not required to provide value for this attribute.

- `fetch` – By default, EclipseLink persistence provider uses a fetch type of `javax.persitence.FetchType.LAZY`: this is a hint to the persistence provider that data should be fetched lazily when it is first accessed (if possible).
  If the default is inappropriate for your application or a particular persistent field, set `fetch` to `FetchType.EAGER`: this is a requirement on the persistence provider runtime that data must be eagerly fetched.

  You are not required to provide value for this attribute.
  For more information, see What You May Need to Know About EclipseLink JPA Lazy Loading.
- `mappedBy` – By default, if the relationship is unidirectional, EclipseLink persistence provider determines the field that owns the relationship.
  If the relationship is bidirectional, then set the `mappedBy` element on the inverse (non-owning) side of the association to the name of the field or property that owns the relationship, as the Usage of @ManyToMany Annotation - Project Class with Generics example shows.

  You are not required to specify the value of this attribute.
- `targetEntity` – By default, if you are using a `Collection` defined using generics, then the persistence provider infers the associated target entity from the type of the object being referenced. Thus, the default is the parameterized type of the `Collection` when defined using generics.
  If your `Collection` does not use generics, then you must specify the entity class that is the target of the association: set the `targetEntity` element on owning side of the association to the `Class` of the entity that is the target of the relationship.

  You are not required to specify the value of this attribute.

The Usage of @ManyToMany Annotation - Employee Class with Generics and Usage of @ManyToMany Annotation - Project Class with Generics examples show how to use this annotation to configure a many-to-many mapping between `Employee` and `Project` using generics.

Usage of @ManyToMany Annotation - Employee Class with Generics

```
@Entity
 public class Employee implements Serializable {
      @Id
      private int id;
      private String name;
      @ManyToMany
      @JoinTable(name="EMP_PROJ",
                  joinColumns=
                        @JoinColumn(name="EMP_ID"),
                  inverseJoinColumns=
                        @JoinColumn(name="PROJ_ID)
      )
```

```
        private Collection<Project> projects;
        ...
  }
```

Note: Use a `@JoinTable` (see Section 9.1.25 "JoinTable Annotation" of the JPA Specification) annotation to define a many-to-many join table; if you do not specify this annotation, EclipseLink will default to `@JoinTable` with the join table name format of `<source-table-name>_<target-table-name>` in uppercase characters, and with columns format of `<source-entity-alias>_<source-primary-key-column>`, `<source-field-name>_ <target-primary-key-column>` (or <source-property-name>_ <target-primary-key-column>) in uppercase characters.

Usage of @ManyToMany Annotation - Project Class with Generics

```
@Entity
 public class Project implements Serializable {
       ...
       @ManyToMany(mappedBy="projects")
       public Set<Employee> getEmployees() {
            return employees;
       }
       ...
 }
```

For more information and examples, see Section 9.1.26 "ManyToMany Annotation" of the JPA Specification.

For more information on EclipseLink direct mappings and relationship mappings, see Relational Mapping Types.

For more information on EclipseLink one-to-one mappings, see Many-to-Many Mapping, and for information on how to configure these mappings, see Configuring a Relational Many-to-Many Mapping.

@MapKey

By default, EclipseLink persistence provider assumes that the primary key of the associated entity is the map key for associations of type `java.util.Map`. If the primary key is a noncomposite primary key annotated with the @Id annotation, an instance of this field or property's type is used as the map key.

Use the `@MapKey` annotation to specify the following:

- some other field or property as the map key if the primary key of the associated entity is not appropriate for your application;

- an embedded composite primary key class (see @EmbeddedId).

The field or property you specify must have a unique constraint (see Section 9.1.4 "UniqueConstraint Annotation" of the JPA Specification).

The `@MapKey` annotation has the following attributes:

- `name` – By default, EclipseLink persistence provider uses the primary key of the associated entity as the `Map` key for a property or field mapped to a `java.util.Map` for noncomposite primary keys, or composite primary keys annotated with the `@IdClass` annotation (see @IdClass).
  If you want to use some other field or property as the map key, set name to the associated entity's `String` field or property name to use.

  You are not required to provide value for this attribute.

In the Project Entity Using @MapKey Annotation example, `Project` owns a one-to-many relationship to instances of `Employee` as a `Map`. The Project Entity Using @MapKey Annotation example shows how to use the `@MapKey` annotation to specify that the key for this `Map` is `Employee` field `empPK`, an embedded composite primary key (see the Employee Entity example) of type `EmployeePK` (see the Project Entity Using @MapKey Annotation example).

Project Entity Using @MapKey Annotation

```
@Entity
 public class Project implements Serializable {
      ...
      @OneToMany(mappedBy="project")
      @MapKey(name="empPK")
      public Map<EmployeePK, Employee> getEmployees() {
           ...
      }
      ...
 }
```

Employee Entity

```
@Entity
 public class Employee implements Serializable {
      ...
      @EmbeddedId
      public EmployeePK getEmpPK() {
           ...
      }

      @ManyToOne
      @JoinColumn(name="proj_id")
      public Project getProject() {
           ...
      }
```

```
        ...
  }
```

EmployeePK Composite Primary Key Class

```
@Embeddable
 public class EmployeePk {

        String name;
        Date birthDate;
        ...
  }
```

For more information and examples, see Section 9.1.27 "MapKey Annotation" of the JPA Specification.

@OrderBy

Use the `@OrderBy` annotation with `@OneToMany` and `@ManyToMany` to specify the following:

- one or more other field or property names to order by;
- different orders (ascending or descending) for each such field or property names.

The `@OrderBy` annotation has the following attributes:

- `value` – By default, EclipseLink persistence provider retrieves the members of an association in ascending order by primary key of the associated entities.
  If you want to order by some other fields or properties and specify different, set `value` to a comma-separated list of the following elements: "`property-or-field-name ASC|DESC`" (see Example 1-65).

  You are not required to provide value for this attribute.

This example shows how to use the `@OrderBy` annotation to specify that the `Project` method `getEmployees` should return a `List` of `Employee` in ascending order by `Employee` field `lastname`, and in descending order by `Employee` field `seniority`.

Project Entity Using @OrderBy Annotation

```
@Entity
 public class Project implements Serializable {
        ...
        @ManyToMany(mappedBy="project")
        @OrderBy("lastname ASC, seniority DESC")
        public List<Employee> getEmployees() {
              ...
        }
        ...
```

```
}
```

For more information and examples, see Section 9.1.28 "OrderBy Annotation" of the JPA Specification.

Mapping Inheritance

By default, EclipseLink persistence provider assumes that all persistent fields are defined by a single entity class.

Use the following annotations if your entity class inherits some or all persistent fields from one or more superclasses:

- @Inheritance
- @MappedSuperclass
- @DiscriminatorColumn
- @DiscriminatorValue

For more information, see Section 2.1.9 "Inheritance" of the JPA Specification.

You can access advanced inheritance options through EclipseLink descriptor API using a DescriptorCustomizer class.

@Inheritance

By default, the EclipseLink persistence provider automatically manages the persistence of entities in an inheritance hierarchy.

Use the `@Inheritance` annotation to customize the persistence provider's inheritance hierarchy support to improve application performance or to match an existing data model.

The `@Inheritance` annotation has the following attributes:

- `strategy` – By default, the EclipseLink persistence provider assumes that all the classes in a hierarchy are mapped to a single table differentiated by the discriminator value (see @DiscriminatorValue) in the table's discriminator column (see @DiscriminatorColumn): `InheritanceType.SINGLE_TABLE`.
  If this is not appropriate for your application or if you must match an existing data model, set `strategy` to the desired `InheritanceType` enumerated type:
  - `SINGLE_TABLE` – all the classes in a hierarchy are mapped to a single table. The table has a discriminator column (@DiscriminatorColumn) whose value (@DiscriminatorValue) identifies the specific subclass to which the instance that is represented by the row belongs.

    Note: This option provides the best support for both polymorphic relationships between entities and queries that range over the class hierarchy. The disadvantages of this option include the need to make nullable columns that should be `NOT NULL`.

For more information, see Section 2.1.10.1 "Single Table per Class Hierarchy Strategy" of the JPA Specification.

- `TABLE_PER_CLASS` – each class is mapped to a separate table. All properties of the class, including inherited properties, are mapped to columns of the table for the class.

Note: This option is available starting in EclipseLink Release 1.1. For earlie versions, you can instead either map each entity subclass independently, o use a `@MappedSuperclass`.

For more information, see Section 2.1.10.2 "Table per Concrete Class Strategy" of the JPA Specification.

- `JOINED` – the root of the class hierarchy is represented by a single table and each subclass is represented by a separate table. Each subclass table contains only those fields that are specific to the subclass (not inherited from its superclass) and primary key columns that serve as foreign keys to the primary keys of the superclass table.
  For more information, see Section 2.1.10.3 "Joined Subclass Strategy" of the JPA Specification.

  You are not required to specify the value of this attribute.

This example shows how to use this annotation to specify that all subclasses of `Customer` will use `InheritanceType.JOINED`. The subclass in the @Inheritance - Subclass Using JOINED example will be mapped to its own table that contains a column for each the persistent properties of `ValuedCustomer` and one foreign key column that contains the primary key to the `Customer` table.

@Inheritance - Root Class Using JOINED

```
import static javax.persistence.InheritanceType.JOINED;
 @Entity
 @Inheritance(strategy=JOINED)
 public class Customer implements Serializable {

      @Id
      private int customerId;
      ...
 }
```

@Inheritance - Subclass Using JOINED

```
@Entity
 public class ValuedCustomer extends Customer {
      ...
 }
```

In the @Inheritance - Root Class Specifying its Discriminator Column example, by default, `InheritanceType.SINGLE_TABLE` applies to `Customer` and all its subclasses. In this example, the default discriminator table column `DTYPE` (@DiscriminatorColumn) is specified as having a discriminator type of `INTEGER` and the `@DiscriminatorValue` for `Customer` is specified as 1. The @Inheritance - Subclass Specifying its Discriminator Value example shows how to specify the discriminator value for subclass

`ValuedCustomer` as 2. In this example, all the persistent properties of both `Customer` and `ValuedCustomer` will be mapped to a single table.

@Inheritance - Root Class Specifying its Discriminator Column

```
@Entity
 @DiscriminatorColumn(discriminatorType=DiscriminatorType.INTEGER)
 @DiscriminatorValue(value="1")
 public class Customer implements Serializable {
      ...
 }
```

@Inheritance - Subclass Specifying its Discriminator Value

```
@Entity
 @DiscriminatorValue(value="2")
 public class ValuedCustomer extends Customer {
      ...
 }
```

For more information, see the following sections of the JPA Specification:

- Section 2.1.9 "Inheritance"
- Section 2.1.10 "Inheritance Mapping Strategies"
- Section 9.1.29 "Inheritance Annotation"

@MappedSuperclass

By default, a EclipseLink persistence provider assumes that all the persistent fields of an entity are defined in that entity.

The `@MappedSuperclass` annotation lets you define mappings in a nonpersistent abstract superclass and enable their inheritance by the subclasses. You can use the @AttributeOverride and @AssociationOverride annotations to override the mapping information in these subclasses.

Use the `@MappedSuperclass` annotation to designate a superclass from which your entity class inherits persistent fields. This is a convenient pattern when multiple entity classes share common persistent fields or properties.

You can annotate this superclass' fields and properties with any of the direct and relationship mapping annotations (such as @Basic and @ManyToMany) as you would for an entity, but these mappings apply only to its subclasses since no table exists for the superclass itself. The inherited persistent fields or properties belong to the subclass' table.

The `@MappedSuperclass` annotation does not have any attributes.

This example shows how to use the `@MappedSuperclass` annotation to specify `Employee` as a mapped superclass. The Extending a Mapped Superclass example shows how to extend this superclass in an entity and how to use the `@AttributeOverride` annotation in the entity class to override configuration made in the superclass.

Usage of the @MappedSuperclass Annotation

```
@MappedSuperclass
 public class Employee implements Serializable {

      @Id
      protected Integer empId;
      @Version
      protected Integer version;

      @ManyToOne
      @JoinColumn(name="ADDR")
      protected Address address;

      ...
 }
```

Extending a Mapped Superclass

```
@Entity
 @AttributeOverride(name="address",
column=@Column(name="ADDR_ID"))
 public class PartTimeEmployee extends Employee {
      @Column(name="WAGE")
      protected Float hourlyWage;

      ...
 }
```

For more information, see the following sections of the JPA Specification:

- Section 9.1.36 "MappedSuperclass Annotation"
- Section 2.1.9.2 "Mapped Superclasses"
- Section 2.1.10 "Inheritance Mapping Strategies"

@DiscriminatorColumn

By default, when @Inheritance attribute strategy is `InheritanceType.SINGLE_TABLE` or `JOINED`, EclipseLink persistence provider creates a discriminator column named `DTYPE` to differentiate classes in an inheritance hierarchy.

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

- specify a discriminator column name if the column name in your data model is not the default column name `DTYPE`;
- specify a discriminator column length that is appropriate for your application or a preexisting data model;
- fine-tune the characteristics of the discriminator column in your database.

The `@DiscriminatorColumn` annotation has the following attributes:

- `columnDefinition` – By default, EclipseLink persistence provider creates a database table column with minimal SQL: empty `String`.
  If you want the column created with more specialized options, set the value of this attribute to the SQL fragment that you want JPA to use when generating the DDL for the column.

  You are not required to specify the value of this attribute.
- `discriminatorType` – By default, EclipseLink persistence provider assumes that the discriminator type is a `String`: `DiscriminatorType.STRING`.
  If you want to use a different type, set the value of this attribute to `DiscriminatorType.CHAR` or `DiscriminatorType.INTEGER`.

  Your @DiscriminatorValue must conform to this type.

  You are not required to specify the value of this attribute.
- `length` – By default, EclipseLink persistence provider assumes that the discriminator column has a maximum length of 255 characters when used to hold a `String` value. Default value of this attribute is 31.
  If this column width is inappropriate for your application or database, set the length to the int value appropriate for your database column.

  Your @DiscriminatorValue must conform to this type.

  You are not required to specify the value of this attribute.
- `name` – By default, EclipseLink persistence provider assumes that the discriminator column is named "`DTYPE`".

  To specify an alternative column name, set name to the `String` column name you want.

  You are not required to specify the value of this attribute.

The @DiscriminatorColumn and @DiscriminatorValue - Root Class example shows how to use this annotation to specify a discriminator column named `DISC` of type `STRING` and length 20. In this example, the @DiscriminatorValue for this class is specified as `CUST`. The subclass in the @DiscriminatorValue - Subclass example specifies its own `@DiscriminatorValue` of `VIP`. In both `Customer` and `ValuedCustomer`, the value for `@DiscriminatorValue` must be convertible to the type specified by `@DiscriminatorColumn` attribute `discriminatorType` and must conform to `@DiscriminatorColumn` attribute `length`.

@DiscriminatorColumn and @DiscriminatorValue - Root Class

```
@Entity
 @Table(name="CUST")
 @Inheritance(strategy=SINGLE_TABLE)
 @DiscriminatorColumn(name="DISC", discriminatorType=STRING,
length=20)
 @DiscriminatorValue(value="CUST")
 public class Customer implements Serializable {
      ...
 }
```

@DiscriminatorValue - Subclass

```
@Entity
```

```
@DiscriminatorValue(value="VIP")
public class ValuedCustomer extends Customer {
     ...
}
```

For more information, see the following sections of the JPA Specification:

- Section 9.1.30 "DiscriminatorColumn Annotation"
- Section 9.1.31 "DiscriminatorValue Annotation"
- Section 2.1.10 "Inheritance Mapping Strategies"

@DiscriminatorValue

By default, when `@Inheritance` attribute strategy is `InheritanceType.SINGLE_TABLE` or `JOINED`, EclipseLink persistence provider uses a `@DiscriminatorColumn` to differentiate classes in the inheritance hierarchy by entity name (see Section 8.1 "Entity" of the JPA Specification).

Use the `@DiscriminatorValue` annotation to specify the discriminator value used to differentiate an entity in this inheritance hierarchy:

- if the entity name is inappropriate for this application;
- to match an existing database schema;

The `@DiscriminatorValue` annotation has the following attributes:

- `value` – Set value to the `String` equivalent of a discriminator value that conforms to the `@DiscriminatorColumn` attributes `discriminatorType` and `length`.

  You are required to specify the value of this attribute.

The @DiscriminatorColumn and @DiscriminatorValue - Root Class example shows how to use this annotation to specify a discriminator column named `DISC` of type `STRING` and length 20. In this example, the `@DiscriminatorValue` for this class is specified as `CUST`. The subclass in the @DiscriminatorValue - Subclass example specifies its own `@DiscriminatorValue` of `VIP`. In both `Customer` and `ValuedCustomer`, the value for `@DiscriminatorValue` must be convertible to the type specified by `@DiscriminatorColumn` attribute `discriminatorType` and must conform to `@DiscriminatorColumn` attribute `length`.

@DiscriminatorColumn and @DiscriminatorValue - Root Class

```
@Entity
 @Table(name="CUST")
 @Inheritance(strategy=SINGLE_TABLE)
 @DiscriminatorColumn(name="DISC", discriminatorType=STRING,
length=20)
 @DiscriminatorValue(value="CUST")
```

```
public class Customer implements Serializable {
      ...
}
```

@DiscriminatorValue - Subclass

```
@Entity
 @DiscriminatorValue(value="VIP")
 public class ValuedCustomer extends Customer {
      ...
 }
```

For more information, see the following sections of the JPA Specification:

- Section 9.1.30 "DiscriminatorColumn Annotation"
- Section 9.1.31 "DiscriminatorValue Annotation"
- Section 2.1.10 "Inheritance Mapping Strategies"

Using Embedded Objects

An embedded object does not have its own persistent identity – it is dependent upon an entity for its identity. For more information, see Section 2.1.5 "Embeddable Classes" of the JPA Specification.

By default, EclipseLink persistence provider assumes that every entity is mapped to its own table. Use the following annotations to override this behavior for entities that are owned by other entities:

- @Embeddable
- @Embedded
- @AttributeOverride
- @AttributeOverrides
- @AssociationOverride
- @AssociationOverrides

For information on EclipseLink aggregate descriptors, refer to Aggregate and Composite Descriptors in Relational Projects; for aggregates advanced configuration options, refer to EclipseLink API.

@Embeddable

Use the `@Embeddable` annotation to specify a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

The `@Embeddable` annotation does not have attributes.

The Usage of the @Embeddable Annotation example shows how to use this annotation to

specify that class `EmploymentPeriod` may be embedded in an entity when used as the type for a persistent field annotated as `@Embedded` (see theUsage of the @Embedded Annotation and @Embedded examples).

Usage of the @Embeddable Annotation

```
@Embeddable
 public class EmploymentPeriod {

      java.util.Date startDate;
      java.util.Date endDate;
      ...
 }
```

For more information, see Section 9.1.34 "Embeddable Annotation" of the JPA Specification.

@Embedded

Use the `@Embedded` annotation to specify a persistent field whose @Embeddable type can be stored as an intrinsic part of the owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the owning entity.

You can use the `@Embedded` annotation in conjunction with `@Embeddable` to model a strict ownership relationship so that if the owning object is removed, the owned object is also removed.

Embedded objects should not be mapped across more than one table.

By default, column definitions (see Section 9.1.5 "Column Annotation" of the JPA Specification) specified in the `@Embeddable` class apply to the `@Embedded` class. If you want to override these column definitions, use the @AttributeOverride annotation.

The `@Embedded` annotation does not have attributes.

The Usage of the @Embedded Annotation example shows how to use this annotation to specify that `@Embeddable` class `EmploymentPeriod` (see the Usage of the @Embeddable Annotation) example may be embedded in the entity class using the specified attribute overrides (@AttributeOverride). If you do not need attribute overrides, you can omit the `@Embedded` annotation entirely: EclipseLink persistence provider will infer that `EmploymentPeriod` is embedded from its `@Embeddable` annotation.

Usage of the @Embedded Annotation

```
@Embeddable
 public class Employee implements Serializable {
      ...
      @Embedded
      @AttributeOverrides({
```

```
        @AttributeOverride(name="startDate",
column=@Column(name="EMP_START")),
        @AttributeOverride(name="endDate",
column=@Column(name="EMP_END"))})
        public EmploymentPeriod getEmploymentPeriod() {
        ...
        }
        ...
  }
```

For more information, see Section 9.1.35 "Embedded Annotation" of the JPA Specification.

@AttributeOverride

By default, EclipseLink persistence provider automatically assumes that a subclass inherits both persistent properties and their basic mappings from the superclass.

Use the `@AttributeOverride` annotation to customize a basic mapping inherited from a @MappedSuperclass or @Embeddable to change the `@Column` (see Section 9.1.5 "Column Annotation" of the JPA Specification) associated with the field or property if the inherited column definition is incorrect for your entity (for example, if the inherited column name is incompatible with a preexisting data model, or invalid as a column name in your database).

If you have more than one `@AttributeOverride` change to make, you must use the @AttributeOverrides annotation.

To customize an association mapping to change its `@JoinColumn` (see Section 9.1.6 "JoinColumn Annotation" of the JPA Specification), use the @AssociationOverride annotation.

The `@AttributeOverride` annotation has the following attributes:

- `column` – The `@Column` that is being mapped to the persistent attribute. The mapping type will remain the same as is defined in the embeddable class or mapped superclass.

  You are required to specify the value of this attribute.
- `name` – The name of the property in the embedded object that is being mapped if property-based access is being used, or the name of the field if field-based access is used.

  You are required to specify the value of this attribute.

The Usage of the @MappedSuperclass Annotation example shows a `@MappedSuperclass` that the entity in the Usage of the @AttributeOverride Annotation example extends. The Usage of the @AttributeOverride Annotation example shows how to use `@AttributeOverride` in the entity subclass to override the `@Column` defined (by default) in the `@MappedSuperclass Employee` for the basic mapping to `Address`.

With the `@AttributeOverride`, the `Employee` table contains the following columns:

- ID
- VERSION
- ADDR_STRING
- WAGE

Without the `@AttributeOverride`, the `Employee` table contains the following columns:

- ID
- VERSION
- ADDRESS
- WAGE

Usage of the @MappedSuperclass Annotation

```
@MappedSuperclass
 public class Employee {

       @Id
       protected Integer id;
       @Version
       protected Integer version;
       protected String address;
       ...
 }
```

Usage of the @AttributeOverride Annotation

```
@Entity
 @AttributeOverride(name="address",
column=@Column(name="ADDR_STRING"))
 public class PartTimeEmployee extends Employee {

       @Column(name="WAGE")
       protected Float hourlyWage;
       ...
 }
```

For more information, see Section 9.1.10 "AttributeOverride Annotation" of the JPA Specification.

@AttributeOverrides

If you need to specify more than one @AttributeOverride, you must specify all your attribute overrides using a single `@AttributeOverrides` annotation.

The `@AttributeOverrides` annotation has the following attributes:

- `value` – To specify two or more attribute overrides, set `value` to an array of `AttributeOverride` instances.

  You are required to specify the value of this attribute.

This example shows how to use this annotation to specify two attribute overrides.

Usage of the @AttributeOverrides Annotation

```
@Entity
 @AttributeOverrides({
      @AttributeOverride(name="address",
column=@Column(name="ADDR_ID")),
      @AttributeOverride(name="id", column=@Column(name="PTID"))
 })
 public class PartTimeEmployee extends Employee {

      @Column(name="WAGE")
      protected Float hourlyWage;

      public PartTimeEmployee() {
      ...
      }

      public Float getHourlyWage() {
      ...
      }

      public void setHourlyWage(Float wage) {
      ...
      }
 }
```

For more information, see Section 9.1.11 "AttributeOverrides Annotation" of the JPA Specification.

@AssociationOverride

By default, EclipseLink persistence provider automatically assumes that a subclass inherits both persistent properties and their association mappings from the superclass.

Use the `@AssociationOverride` annotation to customize an @OneToOne or @ManyToOne mapping inherited from a `@MappedSuperclass` (see @MappedSuperclass) or @Embeddable to change the `@JoinColumn` (see Section 9.1.6 "JoinColumn Annotation" of the JPA Specification) associated with the field or property if the inherited column definition is incorrect for your entity (for example, if the inherited column name is incompatible with a preexisting data model, or invalid as a column name in your database).

If you have more than one `@AssociationOverride` change to make, you must use the @AssociationOverrides annotation.

To customize an association mapping to change its `@Column` (see Section 9.1.5 "Column Annotation" of the JPA Specification), use the `@AttributeOverride` annotation.

The `@AssociationOverride` annotation has the following attributes:

- `joinColumns` – To specify the join columns that are being mapped to the persistent attribute, set the `joinColumns` to an array of `JoinColumn` instances (see Section 9.1.6 "JoinColumn Annotation" of the JPA Specification).

  The mapping type will remain the same as is defined in the embeddable class or mapped superclass.

  You are required to specify the value of this attribute.
- `name` – The name of the property in the embedded object that is being mapped if property-based access is being used, or the name of the field if field-based access is used.

  You are required to specify the value of this attribute.

The Usage of the @MappedSuperclass Annotation example shows a @MappedSuperclass that the entity in the Usage of the @AssociationOverride Annotation example extends. The Usage of the @AssociationOverride Annotation example shows how to use `@AssociationOverride` in the entity subclass to override the `@JoinColumn` defined (by default) in the `@MappedSuperclass Employee` for the association to `Address`.

With the `@AssociationOverride`, the `Employee` table contains the following columns:

- `ID`
- `VERSION`
- `ADDR_ID`
- `WAGE`

Without the `@AssociationOverride`, the `Employee` table contains the following columns:

- `ID`
- `VERSION`
- `ADDRESS`
- `WAGE`

Usage of the @MappedSuperclass Annotation

```
@MappedSuperclass
 public class Employee {

      @Id
      protected Integer id;
      @Version
```

```
        protected Integer version;
        @ManyToOne
        protected Address address;
        ...
  }
```

Usage of the @AssociationOverride Annotation

```
@Entity
 @AssociationOverride(name="address",
joinColumns=@JoinColumn(name="ADDR_ID"))
 public class PartTimeEmployee extends Employee {

        @Column(name="WAGE")
        protected Float hourlyWage;
        ...
  }
```

For more information, see Section 9.1.12 "AssociationOverride Annotation" of the JPA Specification.

@AssociationOverrides

If you need to specify more than one @AssociationOverride, you must specify all your association overrides using a single `@AssociationOverrides` annotation.

The `@AssociationOverrides` annotation has the following attributes:

- `value` – To specify two or more association overrides, set this attribute to an array of `AssociationOverride` instances.

  You are required to specify the value of this attribute.

This example shows how to use this annotation to specify two association overrides.

Usage of the @AssociationOverrides Annotation

```
@Entity
 @AssociationOverrides({
        @AssociationOverride(name="address",
joinColumn=@JoinColumn(name="ADDR_ID")),
        @AssociationOverride(name="phone",
joinColumn=@JoinColumn(name="PHONE_NUM"))
 })
 public class PartTimeEmployee extends Employee {

        @Column(name="WAGE")
        protected Float hourlyWage;
        ...
```

```
}
```

For more information, see Section 9.1.13 "AssociationOverrides Annotation" of the JPA Specification.

Using EclipseLink JPA Extensions

Related Topics

The Java Persistence API (JPA), part of the Java Enterprise Edition 5 (Java EE 5) EJB 3.0 specification, greatly simplifies Java persistence and provides an object relational mapping approach that allows you to declaratively define how to map Java objects to relational database tables in a standard, portable way that works both inside a Java EE 5 application server and outside an EJB container in a Java Standard Edition (Java SE) 5 application.

EclipseLink JPA provides extensions to what is defined in the JPA specification. These extensions come in persistence unit properties, query hints, annotations, EclipseLink own XML metadata, and custom API.

This section explains where and how you use the extensions to customize JPA behavior to meet your application requirements.

For more information, see the following:

- *EclipseLink API Reference*
- *JSR-220 Enterprise JavaBeans v.3.0 Java Persistence API specification*
- *Java EE 5 SDK JPA Javadoc*

Using EclipseLink JPA Extensions for Mapping

EclipseLink defines the following mapping metadata annotations (in addition to JPA-defined ones):

- `@BasicCollection`
- `@BasicMap`
- `@CollectionTable`
- `@PrivateOwned`
- `@JoinFetch`
- `@Mutable`
- `@Transformation`
- `@ReadTransformer`
- `@WriteTransformer`
- `@WriteTransformers`

- @VariableOneToOne

EclipseLink persistence provider searches mapping annotations in the following order:

- @BasicCollection
- @BasicMap
- @EmbeddedId
- @Embedded
- @ManyToMany
- @ManyToOne
- @OneToMany
- @OneToOne

EclipseLink persistence provider applies the first annotation that it finds; it ignores other mapping annotations, if specified. In most cases, EclipseLink does not log warnings or throw exceptions for duplicate or incompatible mapping annotations.

If EclipseLink persistence provider does not find any of the mapping annotations from the preceding list, it applies the defaults defined by the JPA specification: not necessarily the `@Basic` annotation.

How to Use the @BasicCollection Annotation

You can use the `@BasicCollection` annotation to map an `org.eclipse.persistence.mappings.DirectCollectionMapping`, which stores a collection of simple types, such as `String`, `Number`, `Date`, and so on, in a single table. The table must store the value and the foreign key to the source object.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface BasicCollection {
  FetchType fetch() default LAZY;
  Column valueColumn() default @Column;
}
```

Use the `@BasicCollection` annotation in conjunction with a `@CollectionTable` annotation. You can also use it in conjunction with `@Convert` to modify the data value(s) during reading and writing of the collection, as well as with the `@JoinFetch` annotation.

This table lists attributes of the `@BasicCollection` annotation.

Attributes of the @BasicCollection Annotation

Attribute	Description	Default	Required or Optional
`fetch`	The `javax.persistence.FetchType` enumerated type that defines whether	`FetchType.LAZY`	optional

	EclipseLink should lazily load or eagerly fetch the value of the field or property.		
`valueColumn`	The name of the value column (`javax.persistence.Column`) that holds the direct collection data.	`@Column`Note: EclipseLink persistence provider sets the default to the name of the field or property.	optional

Note: If you specify `@BasicCollection` on an attribute of type `Map`, EclipseLink will throw an exception: the type must be `Collection`, `Set` or `List`. If you specify the fetch type as `LAZY`, Collection implementation classes will also not be valid.

This example shows how to use the `@BasicCollection` annotation to specify `Employee` field `responsibilities`.

Usage of the @BasicCollection Annotation

```
@Entity
 public class Employee implements Serializable{
       ...
       @BasicCollection (
            fetch=FetchType.EAGER,
            valueColumn=@Column(name="DESCRIPTION")
       )
       @CollectionTable (
            name="RESPONS",
            primaryKeyJoinColumns=
            {@PrimaryKeyJoinColumn(name="EMPLOYEE_ID",
referencedColumnName="EMP_ID")}
       )
       public Collection getResponsibilities() {
            return responsibilities;
       }
       ...
 }
```

To further customize your mapping, use the DirectCollectionMapping API.

How to Use the @BasicMap Annotation

You can use the `@BasicMap` annotation to map an `org.eclipse.persistence.mappings.DirectMapMapping`, which stores a

collection of key-value pairs of simple types, such as `String`, `Number`, `Date`, and so on, in a single table. The table must store the value and the foreign key to the source object.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface BasicMap {
  FetchType fetch() default LAZY;
  Column keyColumn();
  Convert keyConverter() default @Convert;
  Column valueColumn() default @Column;
  Convert valueConverter() default @Convert;
}
```

Use the `@BasicMap` annotation in conjunction with a `@CollectionTable` annotation, as well as with the `@JoinFetch` annotation.

This table lists attributes of the `@BasicMap` annotation.

Attribute	Description	Default	Requ or Optio
`fetch`	Set this attribute to the `javax.persistence.FetchType` enumerated type to define whether EclipseLink persistence provider has to lazily load or eagerly fetch the value of the field or property.	`FetchType.LAZY`	option
`keyColumn`	Set this attribute to the name of the data column (`javax.persistence.Column`) that holds the direct map key.	`<field-name>_KEY` or `<property-name>_KEY` (in uppercase characters)	option
`keyConverter`	Set this attribute to the key converter (`@Convert`).	`@Convert` – an equivalent of specifying `@Convert("none")` resulting in no converter added to the direct map key.	option
`valueColumn`	Set this attribute to the name of the value column (`javax.persistence.Column`) that holds the direct collection data.	`@Column` Field or property.	option

`valueConverter`	Set this attribute to the value converter (`@Convert`).	`@Convert` – an equivalent of specifying `@Convert("none")` resulting in no converter added to the direct map key.	option

Note: If you specify `@BasicMap` on an attribute of type `Collection`, EclipseLink will throw an exception: the type must be `Map`. If you specify the fetch type as `LAZY`, Map implementation classes are also not valid.

This example shows how to use the `@BasicMap` annotation to specify `Employee` field `licenses`.

Usage of the @BasicMap Annotation

```
@Entity
 @Table(name="CMP3_EMPLOYEE")
 @TypeConverter(
      name="Integer2String",
      dataType=Integer.class,
      objectType=String.class
 )
 public class Employee implements Serializable{
      ...
      @BasicMap (
           fetch=FetchType.EAGER,
           keyColumn=@Column(name="LICENSE"),
           keyConverter=@Convert("licenseConverter"),
           valueColumn=@Column(name="STATUS"),
           valueConverter=@Convert("Integer2String")
      )
      @ObjectTypeConverter(
           name="licenseConverter",
           conversionValues={
                @ConversionValue(dataValue="AL",
objectValue="Alcohol License"),
                @ConversionValue(dataValue="FD",
objectValue="Food License"),
                @ConversionValue(dataValue="SM",
objectValue="Smoking License"),
                @ConversionValue(dataValue="SL",
objectValue="Site Licence")}
      )
      @CollectionTable (
           name="LICENSE",
           primaryKeyJoinColumns={@PrimaryKeyJoinColumn(name="RES
```

```
T_ID")}
      )
      public Map<String, String> getLicenses() {
            return licenses;
      }
      ...
 }
```

To further customize your mapping, use the `DirectMapMapping` API.

How to Use the @CollectionTable Annotation

You can use the `@CollectionTable` annotation in conjunction with a `@BasicCollection` annotation or the `@BasicMap` annotation. If you do not specify the `@CollectionTable`, EclipseLink persistence provider will use the defaults.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface CollectionTable {
  String name() default "";
  String catalog() default "";
  String schema() default "";
  PrimaryKeyJoinColumn[] primaryKeyJoinColumns() default {};
  UniqueConstraint[] uniqueConstraints() default {};
}
```

This table lists attributes of the `@CollectionTable` annotation.

Attribute	Description
`name`	Set this attribute to the `String` name for your collection table.
`catalog`	Set this attribute to the `String` name of the table's catal

`schema`	Set this attribute to the `String` name of the table's sche
`primaryKeyJoinColumns`	Set this attribute to an array of `javax.persistence.PrimaryKeyJoinColumn` instances to specify a primary key column that is used as foreign key to join to another table. If the source entity us a composite primary key, you must specify a primary key column for each field of the composite primary key. If the source entity uses a single primary key, you may choose specify a primary key join column (optional). Otherwise, EclipseLink persistence provider will apply the following defaults: • `javax.persistence.PrimaryKeyJoinColum name` – the same name as the primary key column the primary table of the source entity; • `javax.persistence.PrimaryKeyJoinColum referencedColumnName` – the same name of th primary key column of the primary table of the sou entity. If the source entity uses a composite primary key and yo failed to specify the primary key join columns, EclipseLin will throw an exception.
`uniqueConstraints`	Set this attribute to an array of `javax.persistence.UniqueConstraint` instances you want to place on the table. These constraints are onl used if table generation is in effect.

This example shows how to use the `@CollectionTable` annotation to specify `Employee` field `responsibilities`.

Usage of the @CollectionTable Annotation

```
@Entity
 public class Employee implements Serializable{
       ...
       @BasicCollection (
            fetch="LAZY",
            valueColumn=@Column(name="DESCRIPTION")
       )
       @CollectionTable (
            name="RESPONS",
            primaryKeyJoinColumns=
            {@PrimaryKeyJoinColumn(name="EMPLOYEE_ID",
referencedColumnName="EMP_ID")}
```

```
        )
        public Collection getResponsibilities() {
              return responsibilities;
        }
        ...
  }
```

How to Use the @PrivateOwned Annotation

Use the `@PrivateOwned` annotation in conjunction with a `@OneToOne` annotation, or a `@OneToMany` annotation.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface PrivateOwned {}
```

The `@PrivateOwned` annotation does not have attributes.

This example shows how to use the `@PrivateOwned` annotation to specify `Employee` field `phoneNumbers`.

Usage of the @PrivateOwned Annotation

```
@Entity
 public class Employee implements Serializable {
       ...
       @OneToMany(cascade=ALL, mappedBy="employee")
       @PrivateOwned
       public Collection<PhoneNumber> getPhoneNumbers() {
             return phoneNumbers;
       }
       ...
 }
```

What You May Need to Know About Private Ownership of Objects

When the referenced object is privately owned, the referenced child object cannot exist without the parent object.

When you tell EclipseLink that a relationship is privately owned, you are specifying the following:

- If the source of a privately owned relationship is deleted, then delete the target. Note that this is equivalent of setting a cascade delete.
  For more information, see the following:
  - Optimistic Version Locking Policies and Cascading
  - Section 3.2.2 "Removal" of the JPA Specification
  - Section 3.5.2 "Semantics of the Life Cycle Callback Methods for Entities" of the JPA Specification
  - Section 4.10 "Bulk Update and Delete Operations" of the JPA Specification
  - @OneToOne

- @ManyToOne
- @OneToMany

- If you remove the reference to a target from a source, then delete the target.

Do not configure privately owned relationships to objects that might be shared. An object should not be the target in more than one relationship if it is the target in a privately owned relationship.

The exception to this rule is the case when you have a many-to-many relationship in which a relation object is mapped to a relation table and is referenced through a one-to-many relationship by both the source and the target. In this case, if the one-to-many mapping is configured as privately owned, then when you delete the source, all the association objects will be deleted.

For more information, see How to Use the privateOwnedRelationship Attribute.

How to Use the @JoinFetch Annotation

You can specify the `@JoinFetch` annotation for the following mappings:

- `@OneToOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`
- `@BasicCollection`
- `@BasicMap`

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinFetch {
  JoinFetchType value() default JoinFetchType.INNER;
}
```

Using the `@JoinFetch` annotation, you can enable the joining and reading of the related objects in the same query as the source object.

Note: We recommend setting join fetching at the query level, as not all queries require joining. For more information, see Using Join Reading with ObjectLevelReadQuery.

Alternatively, you can use batch reading, especially for collection relationships. For more information, see Using Batch Reading.

This table lists attributes of the `@JoinFetch` annotation.

Attribute	Description	Defau

`value`	Set this attribute to the `org.eclipse.persistence.annotations.JoinFetchType` enumerated type of the fetch that you will be using. The following are the valid values for the `JoinFetchType`: • `INNER` – This option provides the inner join fetching of the related object. Note: Inner joining does not allow for null or empty values. • `OUTER` – This option provides the outer join fetching of the related object. Note: Outer joining allows for null or empty values. For more information, see the following: • What You May Need to Know About Joins • Using Join Reading with ObjectLevelReadQuery • Configuring Joining at the Mapping Level	`JoinF`

This example shows how to use the `@JoinFetch` annotation to specify `Employee` field `managedEmployees`.

Usage of the @JoinFetch Annotation

```
@Entity
 public class Employee implements Serializable {
       ...
       @OneToMany(cascade=ALL, mappedBy="owner")
       @JoinFetch(value=OUTER)
       public Collection<Employee> getManagedEmployees() {
            return managedEmployees;
       }
       ...
 }
```

How to Use the @Mutable Annotation

You can specify the `@Mutable` annotation for the following mappings:

- `@Basic`
- `@Id`
- `@Version`
- `@Transformation`

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Mutable {
  boolean value() default true;
}
```

Using the `@Mutable` annotation, you can indicate that the value of a complex field itself can be changed or not changed (instead of being replaced).

By default, EclipseLink assumes that `Serializable` types are mutable. In other words, EclipseLink assumes the default `@Mutable(value=true)`.

You can override this configuration using `@Mutable(value=false)`.

By default, EclipseLink assumes that all `@Basic` mapping types, except `Serializable` types, are immutable.

You can override this configuration using `@Mutable(value=true)`. For example, if you need to call `Date` or `Calendar` set methods, you can decorate a `Date` or `Calendar` persistent field using `@Mutable(value=true)`.

Note: For `Date` and `Calendar` types only, you can configure all such persistent fields as mutable by setting global persistence unit property `eclipselink.temporal.mutable` to `true`. For more information, see the EclipseLink JPA Persistence Unit Properties for Mappings table.

Mutable basic mappings affect the overhead of change tracking. Attribute change tracking can only be weaved with immutable mappings.

For more information, see the following:

- Unit of Work and Change Policy
- Using Weaving
- Mutability

This table lists attributes of the `@Mutable` annotation.

Attributes of the @Mutable Annotation

Attribute	Description	Default	Required or Optional
`value`	Set this attribute to one of the following `boolean` values: • `true` – The object is mutable. • `false` – The object is immutable.	`true`	optional

This example shows how to use the `@Mutable` annotation to specify `Employee` field `hireDate`.

Usage of the @Mutable Annotation

```
@Entity
```

```
public class Employee implements Serializable {
      ...
      @Temporal(DATE)
      @Mutable
      public Calendar getHireDate() {
            return hireDate;
      }
      ...
}
```

How to Use the @Transformation Annotation

You can use the `@Transformation` annotation to map an `org.eclipse.persistence.mappings.TransformationMapping`, which allows to map an attribute to one or more database columns.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Transformation {
  FetchType fetch() default EAGER;
  boolean optional() default true;
}
```

Note: The `@Transformation` annotation is optional for transformation mappings: if you specify either a `@ReadTransformer`, `@WriteTransformer` or `@WriteTransformers` annotation, the mapping would become a transformation mapping by default, without you specifying the `@Transformation` annotation.

This table lists attributes of the `@Transformation` annotation.

Attributes of the @Transformation Annotation

Attribute	Description	Default	Required or Optional
`fetch`	The `javax.persistence.FetchType` enumerated type that defines whether EclipseLink should lazily load or eagerly fetch the value of the field or property.	`FetchType.EAGER`	optional
`optional`	Set this attribute to define whether or not the value of the field or property	`true`	optional

	may be null. Note: the value you set is disregarded for primitive types, which are considered mandatory. The following are valid values: • `true` - The value of the field or property may be `null`. • `false` - The value of the field or property may not be `null`.		

For more information, see the following:

- How to Use the @ReadTransformer Annotation
- How to Use the @WriteTransformer Annotation

How to Use the @ReadTransformer Annotation

Use the `@ReadTransformer` annotation with the `@Transformation` mapping to define transformation of one or more database column values into an attribute value. Note that you can only use this annotation if the `@Transformation` mapping is not write-only.

```
@Target(value={METHOD,FIELD})
@Retention(value=RUNTIME)
public @interface ReadTransformer {
  Class transformerClass() default void.class;
  String method() default "";
}
```

This table lists attributes of the `@ReadTransformer` annotation.

Attributes of the @ReadTransformer Annotation

Attribute	Description
`method`	Set this attribute to the String method name that the mapped cla does not assign a value to the attribute; instead, it returns the va attribute.
`transformerClass`	Set this attribute to the `Class` that implements the `org.eclipse.persistence.mappings.transformers.A` interface. This will instantiate the class and use its `buildAttri`

	create the value to be assigned to the attribute. For more information, see see How to Configure Attribute Trans

[1] You must specify either the `transformerClass` or `method`, but not both

How to Use the @WriteTransformer Annotation

Use the `@WriteTransformer` annotation with the `@Transformation` mapping to define transformation of an attribute value to a single database column value. Note that you can only use this annotation if the `@Transformation` mapping is not read-only.

```
@Target(value={METHOD,FIELD})
@Retention(value=RUNTIME)
public @interface WriteTransformer {
  Class transformerClass() default void.class;
  String method() default "";
  Column column() default @Column;
}
```

This table lists attributes of the `@WriteTransformer` annotation.

Attributes of the @WriteTransformer Annotation

Attribute	Description
`method`	Set this attribute to the `String` method name that the mapped method returns the value to be written into the database column Note: for proper support of DDL generation and returning policy return not just an `Object`, but a particular type, as the following `public Time getStartTime()` The method may require a `@Transient` annotation to avoid be `@Basic` by default.
`transformerClass`	Set this attribute to the `Class` that implements the `org.eclipse.persistence.mappings.transformers.F` interface. This will instantiate the class and use its `buildFiel` create the value to be written into the database column. For more information, see see How to Configure Field Transfor Note: for proper support of DDL generation and returning policy

	return not just an `Object`, but a relevant Java type, as the follo `public Time buildFieldValue(Object instance, String fieldName, Session`
`column`	Set this attribute to a `Column` into which the value should be wr You may choose not to set this attribute if a single `WriteTrans` attribute. In this case, the attribute's name will be used as a colu

[1] You must specify either the `transformerClass` or `method`, but not both.

How to Use the @WriteTransformers Annotation

Use the `@WriteTransformers` annotation with the `@Transformation` mapping to wrap multiple `write transformers`. Note that you can only use this annotation if the `@Transformation` mapping is not read-only.

```
@Target(value={METHOD,FIELD})
@Retention(value=RUNTIME)
public @interface WriteTransformers {
  WriteTransformer[] value();
}
```

This table lists attributes of the `@WriteTransformers` annotation.

Attributes of the @WriteTransformers Annotation

Attribute	Description	Default	Required or Optional
`value`	Set this attribute to the array of `WriteTransformer`.	no default	optional

How to Use the @VariableOneToOne Annotation

You can use the `@VariableOneToOne` annotation to map an `org.eclipse.persistence.mappings.VariableOneToOneMapping`, which you use to represent a pointer references between a Java object and an implementer of an interface. This mapping is typically represented by a single pointer (stored in an instance variable) between the source and target objects. In the relational database tables, these mappings are usually implemented using a foreign key and a type code.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface VariableOneToOne {
```

```
  Class targetInterface() default void.class;
  CascadeType[] cascade() default {};
  FetchType fetch() default EAGER;
  boolean optional() default true;
  DiscriminatorColumn discriminatorColumn() default
@DiscriminatorColumn;
  DiscriminatorClass[] discriminatorClasses() default {};
}
```

Specify the `@VariableOneToOne` annotation within an `@Entity` (see Section 8.1 "Entity" of the JPA Specification), `@MappedSuperclass` or `@Embeddable` class.

This table lists attributes of the `@VariableOneToOne` annotation.

Attributes of the @VariableOneToOneAnnotation

Attribute	Description
`cascade`	Set this attribute to an array of `javax.persistence.Ca` objects to indicate operations that must be cascaded to th association.
`discriminatorClasses`	Set this attribute to an array of `org.eclipse.persistence.annotations.Discrim` objects to provide a list of discriminator types that can be u variable one-to-one mapping. If none are specified, then those entities within the persiste implement the target interface will be added to the list of ty the discriminator type will default as follows: • If the `javax.persistence.DiscriminatorCol` `STRING`, it is `Entity.name()` • If the `DiscriminatorColumn` type is `CHAR`, it is th `Entity` class • If the `DiscriminatorColumn` type is `INTEGER` it after the highest integer explicitly added.
`discriminatorColumn`	Set this attribute to the `javax.persistence.Discrimi` that will hold the type indicators. If the discriminator column is not specified, the name of th column defaults to `"DTYPE"`, and the discriminator type -
`fetch`	The `javax.persistence.FetchType` enumerated type whether EclipseLink should lazily load or eagerly fetch the or property.
`optional`	Set this attribute to define whether or not the association is

	The following are valid values: • `true` - The association is optional. • `false` - A non-null relationship must always exist.
`targetInterface`	Set this attribute to an interface class that is the target of t If not specified, it will be inferred from the type of the objec referenced.

How to Use the Persistence Unit Properties for Mappings

This table lists the persistence unit properties that you can define in a `persistence.xml` file to configure EclipseLink mappings.

EclipseLink JPA Persistence Unit Properties for Mappings

Property	Usage

`eclipselink.temporal.mutable`	Specify whether or not EclipseLink JPA should `Calendar` persistent fields as mutable objects
	
	The following are the valid values:
	• `true` – all `Date` and `Calendar` persiste
	• `false` – all `Date` and `Calendar` persis immutable.
	
	For more information, see the following:
	• How to Use the @Mutable Annotation
	• Mutability
	
	Example: `persistence.xml` file
	`<property name="eclipselink.temporal.mutable" value`
	
	Example: property `Map`
	`import org.eclipse.persistence.config.PersistenceUn`
	`propertiesMap.put(PersistenceUnitProperties.TEMPORA`

For information about the use of annotations as opposed to persistence unit properties and vice versa, see the following:

- What you May Need to Know About Overriding Annotations in JPA

- What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

For more information about persistence unit properties, see What you May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

Using EclipseLink JPA Converters

EclipseLink defines the following converter annotations (in addition to JPA-defined ones):

- `@Converter`
- `@TypeConverter`
- `@ObjectTypeConverter`
- `@StructConverter`
- `@Convert`

EclipseLink persistence provider searches the converter annotations in the following order:

- `@Convert`
- `@Enumerated`
- `@Lob`
- `@Temporal`
- Serialized (automatic)

You can define converters at the class, field and property level. You can specify EclipseLink converters on the following classes:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification)
- `@MappedSuperclass`
- `@Embeddable`

You can use EclipseLink converters with the following mappings:

- `@Basic`
- `@Id`
- `@Version`
- `@BasicMap`
- `@BasicCollection`

If you specify a converter with any other type of mapping annotation, EclipseLink will throw an exception.

How to Use the @Converter Annotation

You can use `@Converter` annotation to specify a custom converter for modification of the data value(s) during the reading and writing of a mapped attribute.

```
@Target({TYPE, METHOD, FIELD})
```

```
@Retention(RUNTIME)
public @interface Converter {
  String name();
  Class converterClass();
}
```

This table lists attributes of the `@Converter` annotation.

Attribute	Description
`name`	Set this attribute to the `String` name for your converter. Ensure th name is unique across the persistence unit
`converterClass`	Set this attribute to the `Class` of your converter. This class must in the EclipseLink `org.eclipse.persistence.mappings.converters.Conve` interface.

This example shows how to use the `@Converter` annotation to specify `Employee` field `gender`.

Usage of the @Converter Annotation

```
@Entity
 public class Employee implements Serializable{
       ...
       @Basic
       @Converter (
            name="genderConverter",
            converterClass=org.myorg.converters.GenderConverter.cl
ass
       )
       @Convert("genderConverter")
       public String getGender() {
            return gender;
       }
       ...

}
```

How to Use the @TypeConverter Annotation

The `@TypeConverter` is an EclipseLink-specific annotation. You can use it to specify an `org.eclipse.persistence.mappings.converters.TypeConversionConverte r` for modification of the data value(s) during the reading and writing of a mapped attribute.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface TypeConverter {
  String name();
  Class dataType() default void.class;
  Class objectType() default void.class;
}
```

This table lists attributes of the `@TypeConverter` annotation.

Attribute	Description	Default	Required/Optional
`name`	Set this attribute to the `String` name for your converter. Ensure that this name is unique across the persistence unit.	no default	required
`dataType`	Set this attribute to the type stored in the database.	`void.class`[1]	optional
`objectType`	Set the value of this attribute to the type stored on the entity.	`void.class`[1]	optional

[1] The default is inferred from the type of the persistence field or property.

This example shows how to use the `@TypeConverter` annotation to convert the `Double` value stored in the database to a `Float` value stored in the entity.

Usage of the @TypeConverter Annotation

```
@Entity
 public class Employee implements Serializable{
      ...
      @TypeConverter (
           name="doubleToFloat",
           dataType=Double.class,
           objectType=Float.class,
      )
      @Convert("doubleToFloat")
      public Number getGradePointAverage() {
           return gradePointAverage;
      }
      ...
 }
```

How to Use the @ObjectTypeConverter Annotation

You can use the `@ObjectTypeConverter` annotation to specify 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.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface ObjectTypeConverter {
  String name();
  Class dataType() default void.class;
  Class objectType() default void.class;
  ConversionValue[] conversionValues();
  String defaultObjectValue() default "";
}
```

This table lists attributes of the `@ObjectTypeConverter` annotation.

Attribute	Description	Default	Requ Optio
`name`	Set this attribute to the `String` name for your converter. Ensure that this name is unique across the persistence unit	no default	requir
`dataType`	Set this attribute to the type stored in the database.	`void.class`[1]	option
`objectType`	Set the value of this attribute to the type stored on the entity.	`void.class`[1]	option
`conversionValues`	Set the value of this attribute to the array of conversion values (instances of `ConversionValue`: `String objectValue` and `String dataValue`. See the Usage of the @ObjectTypeConverter Annotation example, to be used with the object converter.	no default	requir
`defaultObjectValue`	Set the value of this attribute to the default object value. Note that this argument is for dealing with legacy data if the data value is missing.	empty `String`	option

[1] The default is inferred from the type of the persistence field or property.

This example shows how to use the `@ObjectTypeConverter` annotation to specify the `Employee` field `gender`.

Usage of the @ObjectTypeConverter Annotation

```
public class Employee implements Serializable{
       ...
       @ObjectTypeConverter (
            name="genderConverter",
            dataType=java.lang.String.class,
            objectType=java.lang.String.class,
            conversionValues={
                 @ConversionValue(dataValue="F",
objectValue="Female"),
                 @ConversionValue(dataValue="M",
objectValue="Male")}
       )
       @Convert("genderConverter")
       public String getGender() {
            return gender;
       }
       ...
  }
```

How to Use the @StructConverter Annotation

The `@StructConverter` is an EclipseLink-specific annotation. You can add it to an `org.eclipse.persistence.platform.database.DatabasePlatform` using its `addStructConverter` method to enable custom processing of `java.sql.Struct` types.

```
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface StructConverter {
  String name();
  String converter();
}
```

This table lists attributes of the `@StructConverter` annotation.

Attribute	Description
`name`	Set this attribute to the `String` name for your converter. Ensure that this name is unique across the persistence unit.

converter	Set this attribute to the converter class as a String. This class must implement the EclipseLink org.eclipse.persistence.mappings.converters.Converter interface.

This example shows how to define the @StructConverter.

Defining the @StructConverter

```
@StructConverter(name="MyType",
       converter="myproject.converters.MyStructConverter")
```

You can specify the @StructConverter annotation anywhere in an Entity with the scope being the whole session.

EclipseLink will throw an exception if you add more than one StructConverter that affects the same Java type.

A @StructConverter exists in the same namespaces as @Converter. EclipseLink will throw a validation exception if you add a Converter and a StructConverter of the same name.

Note: You can also configure structure converters in a sessions.xml file (see What You May Need to Know About EclipseLink JPA Overriding Mechanisms).

Using Structure Converters to Configure Mappings

In EclipseLink, a DatabasePlatform (see Database Platforms) holds a structure converter. An org.eclipse.persistence.database.platform.converters.StructConverter affects all objects of a particular type read into the Session that has that DatabasePlatform. This prevents you from configuring the StructConverter on a mapping-by-mapping basis. To configure mappings that use the StructConverter, you call their setFieldType(java.sql.Types.STRUCT) method. You must call this method on all mappings that the StructConverter will affect – if you do not call it, errors might occur.

The JPA specification requires all @Basic mappings (see @Basic) that map to a non-primitive or a non-primitive-wrapper type have a serialized converter added to them. This enables certain STRUCT types to map to a field without serialization.

You can use the existing @Convert annotation with its value attribute set to the StructConverter name – in this case, EclipseLink will apply appropriate settings to the mapping. This setting will be required on all mappings that use a type for which a StructConverter has been defined. Failing to configure the mapping with the @Convert will cause an error.

For more information, see the following:

- Object-Relational Data Type Structure Mapping
- Object-Relational Data Type Descriptors

How to Use the @Convert Annotation

The `@Convert` annotation specifies that a named converter should be used with the corresponding mapped attribute.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Convert {
  String value() default "none";
}
```

The `@Convert` has the following reserved names:

- `serialized` – places the `org.eclipse.persistence.mappings.converters.SerializedObjectConverter` on the associated mapping.
- `none` – does not place a converter on the associated mapping.

This table lists attributes of the `@Convert` annotation.

Attributes of the @Convert Annotation

Attribute	Description	Default	Required or Optional
`value`	Set this attribute to the `String` name for your converter.	`"none"` `String`	optional

This example shows how to use the `@Convert` annotation to define the `Employee` field `gender`.

Usage of the @Convert Annotation

```
@Entity
 @Table(name="EMPLOYEE")
 @Converter(
      name="genderConverter",
           converterClass=org.myorg.converters.GenderConverter.class
 )
 public class Employee implements Serializable{
      ...
      @Basic
```

```
        @Convert("genderConverter")
        public String getGender() {
             return gender;
        }
        ...
  }
```

Using EclipseLink JPA Extensions for Entity Caching

The EclipseLink cache is an in-memory repository that stores recently read or written objects based on class and primary key values. EclipseLink uses the cache to do the following:

- Improve performance by holding recently read or written objects and accessing them in-memory to minimize database access.
- Manage locking and isolation level.
- Manage object identity.

For more information about the EclipseLink cache and its default behavior, see Introduction to Cache.

EclipseLink defines the following entity caching annotations:

- `@Cache`
- `@TimeOfDay`
- `@ExistenceChecking`

EclipseLink also provides a number of persistence unit properties that you can specify to configure the EclipseLink cache (see How to Use the Persistence Unit Properties for Caching). These properties may compliment or provide an alternative to the usage of annotations.

For more information, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties

How to Use the @Cache Annotation

EclipseLink uses identity maps to cache objects in order to enhance performance, as well as maintain object identity. You can control the cache and its behavior by decorating your entity classes with the `@Cache` annotation.

```
@Target({TYPE})
 @Retention(RUNTIME)
 public @interface Cache {
```

```
      CacheType type() default SOFT_WEAK;
      int size() default 100;
      boolean shared() default true;
      int expiry() default -1;
      TimeOfDay expiryTimeOfDay() default
@TimeOfDay(specified=false);
      boolean alwaysRefresh() default false;
      boolean refreshOnlyIfNewer() default false;
      boolean disableHits() default false;
      CacheCoordinationType coordinationType() default
SEND_OBJECT_CHANGES;
  }
```

You may define the `@Cache` annotation on the following:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification);
- `@MappedSuperclass`;
- the root of the inheritance hierarchy (if applicable).

Note: If you define the `@Cache` annotation on an inheritance subclass, the annotation will be ignored.

Attributes of the @Cache Annotation

Attribute	Description

`type`	Set this attribute to the type (`org.eclipse.persistence.annotations.CacheTyp` of the cache that you will be using. The following are the valid values for the `CacheType`: • `FULL` – This option provides full caching and guarante objects are cached and not removed. Note: this process may be memory-intensive when ma • `WEAK` – This option is similar to `FULL`, except that obje using weak references. This option uses less memory complete garbage collection and provides full caching identity. We recommend using this identity map for tra started, stay on the server side. • `SOFT` – This option is similar to `WEAK` except that the m objects using soft references. This identity map enable collection when memory is low. It provides full caching identity. • `SOFT_WEAK` – This option is similar to `WEAK` except tha most frequently used subcache that uses soft referenc subcache is proportional to the size of the identity map uses soft references to ensure that these objects are g only if the system is low on memory. We recommend u map in most circumstances as a means to control mem cache. • `HARD_WEAK` – This option is similar to `SOFT_WEAK` exc a most frequently used subcache that uses hard refere identity map if soft references are not suitable for your • `CACHE` – With this option, a cache identity map mainta of objects that you specify in your application. Objects the cache on a least-recently-used basis. This option a for the most commonly used objects. Note: this option furnishes caching and identity, but do identity. • `NONE` – This option does not preserve object identity a objects. We do not recommend using this option.
`size`	Set this attribute to an `int` value to define the size of cache t objects).
`shared`	Set this attribute to a `boolean` value to indicate whether cac should be in the shared cache or in a client isolated cache (s Session Cache). The following are the valid values: • `true` - use shared cache for cached instances; • `false` - use client isolated cache for cached instances

`expiry`	Set this attribute to the `int` value to enable the expiration of after a fixed period of time (milliseconds). Queries executed after this will be forced back to the database for a refreshed
`expiryTimeOfDay`	Set this attribute to a specific time of day (`org.eclipse.persistence.annotations.TimeOfDa` instance will expire. Queries executed against the cache afte back to the database for a refreshed copy.
`alwaysRefresh`	Set this attribute to a `boolean` value of `true` to force all que database to always refresh the cache.
`refreshOnlyIfNewer`	Set this attribute to a `boolean` value of `true` to force all que database to refresh the cache only if the data received from t query is newer than the data in the cache (as determined by locking field). Note: This option only applies if one of the other refreshing o `alwaysRefresh`, is already enabled. Note: A version field is necessary to apply this feature. For more information, see the following: • What You May Need to Know About Version Fields • Optimistic Version Locking Policies • Configuring Locking • Section 3.4 "Optimistic Locking and Concurrency" of th • Section 9.1.17 "Version Annotation" of the JPA Specifi
`disableHits`	Set this attribute to a `boolean` value of `true` to force all que cache for hits, but still resolve against the cache for identity. queries to hit the database.
`coordinationType`	Set this attribute to the cache coordination mode (`org.eclipse.persistence.annotations.CacheCoo` enumerated type). The following are the valid values for the `CacheCoordinati` • `SEND_OBJECT_CHANGES` – This option sends a list of (including information about the changes). This data is receiving cache. • `INVALIDATE_CHANGED_OBJECTS` – This option send identities of the objects that have changed. The receiv invalidates the objects (rather than changing any of the • `SEND_NEW_OBJECTS_WITH_CHANGES` – This option i `SEND_OBJECT_CHANGES` except it also includes any n objects from the transaction.

	• `NONE` – This option does not coordinate cache. For more information, see Cache Coordination.

Note: If you define the `@Cache` annotation on `@Embeddable` (see @Embeddable), EclipseLink will throw an exception.

This example shows how to achieve the desired behavior of the EclipseLink cache by defining the attributes of the `@Cache` annotation.

Usage of @Cache Annotation

```
@Entity
 @Table(name="EMPLOYEE")
 @Cache (
      type=CacheType.WEAK,
      isolated=false,
      expiry=600000,
      alwaysRefresh=true,
      disableHits=true,
      coordinationType=INVALIDATE_CHANGED_OBJECTS
      )
 public class Employee implements Serializable {
      ...
 }
```

What You May Need to Know About Version Fields

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

Use the `@Version` annotation (see Configuring Locking) 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 (see Section 9.1.1 "Table Annotation" of the JPA Specification);
- your application does not modify the version property or field.

How to Use the Persistence Unit Properties for Caching

The EclipseLink JPA Properties for Caching table lists the persistence unit properties that

you can define in a `persistence.xml` file to configure the EclipseLink cache.

For more information, see the following:

- Introduction to Cache
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms
- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties

EclipseLink JPA Properties for Caching

Property	Usage
`eclipselink.cache.type.default`	The default type of session cache. A session cache is a shared cache that s When you read objects from or write obje session, EclipseLink saves a copy of the cache and makes them accessible to chil an `EntityManagerFactory` wraps an `org.eclipse.persistence.sessio` managers wrap an `org.eclipse.pers` `org.eclipse.persistence.sessio` information about sessions, see Introduct The following are the valid values for the the `org.eclipse.persistence.con` • `Full` – This option provides full ca never flushed from memory unless For more information, see Full Ide • `Weak` – This option is similar to `Fu` using weak references. This option not provide a durable caching strat recommend using this identity map the server side. For more information, see Weak Id • `Soft` – This option is similar to `We` using soft references. This identity memory is low. It provides full cach For more information, see Soft Ide • `SoftWeak` – This option is similar frequently used subcache that use this identity map in most circumsta by the cache. For more information, see Soft Ca Weak Identity Map. • `HardWeak` – This option is similar frequently used subcache that use For more information, see Soft Ca Weak Identity Map.

	• `NONE` – This option does not prese objects. Oracle does not recomme For more information, see No Iden **"eclipselink.cache.shared"="fals**
	
	Note: The values are case-sensitive.
	
	Note: Using this property, you can overri the @Cache Annotation) attribute type.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.cache.type.defaul`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.CacheTy`
	`import org.eclipse.persistence.config.Persist`
	`propertiesMap.put(PersistenceUnitProperties.C`

`eclipselink.cache.size.default`	The default maximum number of objects
	
	Valid values: 0 to `Integer.MAX_VALUE`
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.cache.size.defaul`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persist`
	`propertiesMap.put(PersistenceUnitProperties.C`

`eclipselink.cache.shared.default`	The default for whether or not the Eclipse client sessions. The following are the valid values: • `true` – The session cache service you read objects from or write obje session, EclipseLink saves a copy session's cache and makes them a session. • `false` – The session cache servic isolated client can reference objec can reference objects in the isolate **Example**: `persistence.xml` file

	`<property name="eclipselink.cache.shared.defa`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persist`
	`propertiesMap.put(PersistenceUnitProperties.C`

`eclipselink.cache.type.<ENTITY>`	The type of session cache for the JPA en
	name `<ENTITY>`.
	For more information on entity names, se
	Specification.
	
	The following are the valid values for the
	the `org.eclipse.persistence.con`
	
	• "`Full`" – see `eclipselink.cac`
	• "`HardWeak`" – see `eclipselink`
	• "`NONE`" – see `eclipselink.cac`
	• "`SoftWeak`" – see `eclipselink`
	• "`Weak`" – see `eclipselink.cac`
	Note: Using this property, you can
	`type`.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.cache.type.Order"`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.CacheTy`
	`import org.eclipse.persistence.config.Persist`
	`propertiesMap.put(PersistenceUnitProperties.C`

`eclipselink.cache.size.<ENTITY>`	The maximum number of JPA entities of
	`<ENTITY>` allowed in an EclipseLink cac
	see Section 8.1 "Entity" of the JPA Speci
	
	Valid values: 0 to `Integer.MAX_VALUE`
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.cache.size.Order"`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persist`
	`propertiesMap.put(PersistenceUnitProperties.C`

`eclipselink.cache.shared.<ENTITY>`	Whether or not the EclipseLink session c for JPA entities of the type denoted by JP

	For more information on entity names, se
	Specification.
	
	The following are the valid values:
	
	• `true` – The session cache service
	you read objects from or write obje
	session, EclipseLink saves a copy
	session's cache and makes them a
	session.
	• `false` – The session cache servic
	isolated client can reference objec
	can reference objects in the isolate
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.cache.shared.Orde`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persist`
	`propertiesMap.put(PersistenceUnitProperties.C`

`eclipselink.flush-clear.cache`	Defines the `EntityManager` cache beha followed by a call to the `clear` method. Y either an `EntityManagerFactory` (eith `createEntityManagerFactory` meth `EntityManager` (in the map passed to t Note that the latter overrides the former. The following are the valid values for the the `org.eclipse.persistence.con` • `Drop` – The call to the `clear` meth `EntityManager`'s cache. This me memory. However, after commit th stale data. • `DropInvalidate` – Even though drop of the entire `EntityManage` object updated or deleted are inva This mode is slower than `Drop`, bu prevents stale data. • `Merge` – The call to the `clear` me `EntityManager`'s cache of objec leaves the shared cache in a perfe least memory-efficient mode; the m transaction. **Example**: `persistence.xml` file

	`<property name="eclipselink.flush-clear.cache`
	Example: property `Map`
	`import org.eclipse.persistence.config.Persist` `propertiesMap.put(PersistenceUnitProperties.F`

How to Use the @TimeOfDay Annotation

You can use the `@TimeOfDay` annotation to specify a time of day using a `Calendar` instance. By doing so, you configure cache expiry on an entity class.

```
@Target({})
@Retention(RUNTIME)
public @interface TimeOfDay {
  int hour() default 0;
  int minute() default 0;
  int second() default 0;
  int millisecond() default 0;
}
```

This table lists attributes of the `@TimeOfDay` annotation.

Attributes of the @TimeOfDay Annotation

Attribute	Description	Default	Required o Optional
`hour`	Set this attribute to the `int` value representing an hour of the day.	0	optional
`minute`	Set this attribute to the `int` value representing a minute of the day.	0	optional
`second`	Set this attribute to the `int` value representing a second of the day.	0	optional
`millisecond`	Set this attribute to the `int` value representing a millisecond of the day.	0	optional

How to Use the @ExistenceChecking Annotation

Use the `@ExistenceChecking` annotation to specify the type of checking that EclipseLink should use when determining if an `Entity` is new or already existing.

On a merge operation, this annotation determines whether or not EclipseLink should only use the cache to check if an object exists, or should read the object (from the database or cache).

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ExistenceChecking {
  ExistenceType value() default CHECK_CACHE;
}
```

You may define the `@ExistenceChecking` annotation on the following:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification);
- `@MappedSuperclass`;

This table lists attributes of the `@ExistenceChecking` annotation.

Attributes of the @ExistenceChecking Annotation

Attribute	Description	Defau
`value`	Set this attribute to the type (`org.eclipse.persistence.annotations.ExistenceType` enumerated type) of the existence checking that you will be using to determine if an insert or an update operation should occur for an object. The following are the valid values for the `ExistenceType`: • `CHECK_CACHE` – This option assumes that if the object's primary key does not include `null` and it is in the cache, then this object must exist. • `CHECK_DATABASE` – This option triggers the object existence check on a database. • `ASSUME_EXISTENCE` – This option assumes that if the object's primary key does not include `null`, then the object must exist. You may choose this option if your application guarantees the existence checking, or is not concerned about it. • `ASSUME_NON_EXISTENCE` – This option assumes that the object does not exist. You may choose this option if your application guarantees the existence checking, or is not concerned about it. If you specify this option, EclipseLink will force the call of an insert operation.	One o applie • •

Using EclipseLink JPA Extensions for Customization and Optimization

EclipseLink defines one descriptor customizer annotation – `@Customizer` (see How to Use the @Customizer Annotation).

EclipseLink also provides a number of persistence unit properties that you can specify to configure EclipseLink customization and validation (see How to Use the Persistence Unit Properties for Customization and Validation). These properties may compliment or provide an alternative to the usage of annotations.

Note: Persistence unit properties always override the corresponding annotations' attributes.

For more information, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties

In addition, EclipseLink provides a persistence unit property that you can specify to optimize your application (see How to Use the Persistence Unit Properties for Optimization).

How to Use the @Customizer Annotation

Use the `@Customizer` annotation to specify a class that implements the `org.eclipse.persistence.config.DescriptorCustomizer` interface and that is to be run against a class' descriptor after all metadata processing has been completed. See `eclipselink.descriptor.customizer.<ENTITY>` for more information.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface Customizer {
  Class value();
}
```

You can define the `@Customizer` annotation on the following:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification);
- `@MappedSuperclass`
- `@Embeddable`

Note: A `@Customizer` is not inherited from its parent classes.

This table lists attributes of the `@Customizer` annotation.

Attributes of the @Customizer Annotation

Attribute	Description	Default	Required / Optional
`value`	Set this attribute to the `Class` of the descriptor customizer that you want to apply to your entity's descriptor.	no default	required

This example shows how to use the `@Customizer` annotation.

Usage of the @Customizer Annotation

```
@Entity
 @Table(name="EMPLOYEE")
 @Customizer(mypackage.MyCustomizer.class)
 public class Employee implements Serializable {
      ...
 }
```

How to Use the Persistence Unit Properties for Customization and Validation

This table lists the persistence unit properties that you can define in a `persistence.xml` file to configure EclipseLink customization and validation.

EclipseLink JPA Properties for Customization and Validation

Property	Usage

`eclipselink.orm.throw.exceptions`	Specify whether or not EclipseLink JP with any of the files listed in a `persi`
	
	The following are the valid values:
	
	• `true` – throw exceptions.
	• `false` – log warning only.
	
	Example: `persistence.xml` file
	
	`<property name="oracle.orm.throw.exceptio`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.exception-handler`	Specify an EclipseLink exception han
	`org.eclipse.persistence.exc`
	argument) constructor. Use this class
	`java.lang.RuntimeException`,
	database operation.
	
	For more information, see Exception
	
	Valid values: class name of an `Exce`
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.exception-han`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.weaving`	Control whether or not the weaving o
	uses weaving to enhance JPA entitie
	optimizations.
	
	The following are the valid values:
	
	• `true` – weave entity classes c
	• `false` – do not weave entity c
	• `static` – weave entity classe
	
	This assumes that classes have alre
	deploying them.
	
	For more information, see the followi
	
	• Using EclipseLink JPA Weavi
	• How to Configure Dynamic We
	• How to Configure Static Weav
	
	Example: `persistence.xml`
	
	`<property name="eclipselink.weaving" valu`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.weaving.lazy`	Enable or disable the lazy one-to-one
	
	The following are the valid values:
	
	• `true` – enable lazy one-to-one
	• `false` – disable lazy one-to-o
	
	Note: you may set this option only if
	`eclipselink.weaving.lazy` opti
	
	For more information, see the followi
	
	• Using EclipseLink JPA Weavi
	• What You May Need to Know
	
	Example: `persistence.xml`
	
	`<property name="eclipselink.weaving.lazy`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.weaving.changetracking`	Enable or disable the `AttributeLe` The following are the valid values: • `true` – enable the `Attribut` all mappings allowing change • `false` – disable the `Attribu` applies: • you cannot weave at all • you do not want your cl • you wish to disable this the `java.util.Date` underlying instance var **Note**: you may set this option only if `eclipselink.weaving.changet` For more information, see the followi • Using EclipseLink JPA Weavi • Configuring Change Policy **Example**: `persistence.xml`

	`<property name="eclipselink.weaving.chan`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.weaving.fetchgroups`	Enable or disable fetch groups throu
	
	The following are the valid values:
	
	• `true` – enable the use of fetch
	• `false` – disable the use of fe
	  • you cannot weave at al
	  • you do not want your cl
	  you wish to disable this
	
	Note: you may set this option only if
	`eclipselink.weaving.fetchgr`
	
	For more information, see the followi
	
	• Using EclipseLink JPA Weavi
	• Configuring Fetch Groups
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.weaving.fetc`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.weaving.internal`	Enable or disable internal optimizatio
	
	The following are the valid values:
	
	• `true` – enable internal optimiz
	• `false` – disable internal optim
	
	Note: you may set this option only if
	`eclipselink.weaving.interna`
	
	For more information, see Using Ecli
	
	Example: `persistence.xml`
	
	`<property name="eclipselink.weaving.inte`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.weaving.eager`	Enable or disable indirection on eage
	
	The following are the valid values:
	
	• `true` – enable indirection on e
	• `false` – disable indirection or
	
	Note: you may set this option only if
	`eclipselink.weaving.eager` op
	
	For more information, see the followi
	
	• Using EclipseLink JPA Weavi
	• Value Holder Indirection
	
	Example: `persistence.xml`
	
	`<property name="eclipselink.weaving.eage`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.session.customizer`	Specify an EclipseLink session custo
	`org.eclipse.persistence.con`
	constructor. Use this class' `customi`
	`org.eclipse.persistence.ses`
	API.
	
	For more information, see Session C
	
	Valid values: class name of a `Sessi`
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.session.custo`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.descriptor.customizer.<ENTITY>`	Specify an EclipseLink descriptor cus
	`org.eclipse.persistence.con`
	argument) constructor. Use this class
	`org.eclipse.persistence.des`
	EclipseLink descriptor and mapping A
	
	For more information on entity names
	
	For more information, see Descriptor
	
	Note: EclipseLink does not support m
	`DescriptorCustomizer` class fully
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.descriptor.cu`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.validation-only`	Specify whether or not deployment s The following are the valid values: • `true` – deployment is only for • `false` – normal deployment; t

	Example: `persistence.xml` file
	`<property name="eclipselink.validation-o`
	Example: property `Map`
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

`eclipselink.classloader`	Specify the class loader to use for cr
	`createEntityManagerFactory` m
	Example: `persistence.xml` file
	`<property name="eclipselink.classloader"`
	Example: property `Map`
	`import org.eclipse.persistence.config.Pe`
	`propertiesMap.put(PersistenceUnitPropert`

For information about the use of annotations as opposed to persistence unit properties and vice versa, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms

For more information about persistence unit properties, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

How to Use the Persistence Unit Properties for Optimization

This table lists the persistence unit properties that you can define in a `persistence.xml` file to optimize your EclipseLink application.

EclipseLink JPA Persistence Unit Properties for Optimization

Property	Usage

`eclipselink.profiler`	The type of the performanc
	For more information on pe
	The following are the valid
	`org.eclipse.persiste`
	• `PerformanceProf`
	`(org.eclipse.pe`

	information, see Mea
	• `QueryMonitor` – M (`org.eclipse.pe` a simple low-overhe may want to use this
	• `NoProfiler` – Do r
	• Custom profiler – Us `org.eclipse.per` argument constructo
	
	Example: `persistence.`
	
	`<property name="eclipselink.pr`
	
	Example: property `Map`
	
	`import org.eclipse.persistence`
	`import org.eclipse.persistence`
	`propertiesMap.put(PersistenceU`
	
	Example: `persistence.`
	
	`<property name="eclipselink.pr`
	
	Note: Ensure that `MyProf`
	
	Example: property `Map`
	
	`import org.eclipse.persistence`
	`import org.eclipse.persistence`
	`propertiesMap.put(PersistenceU`

`eclipselink.persistence.context.reference-mode`	Specify whether or not the context using Java's weak In cases where your applic persistence context, use th from the persistence conte You can set this property e call, or globally in the `pers` The following are the valid `org.eclipse.persiste` • `HARD` - Use this opti These objects will n persistence context • `WEAK` - Use this opti tracking (see Attribu object no longer refe change is made to a be available for garb New and removed o

will also be held by h

- `FORCE_WEAK` - Use
are to be held by we
is moved to a hard r
However, any object
collected before thei
changes.
New and removed o
collection.

Using Java, you can config
`acquireUnitOfWork(Re`
default, use the `Session`'s

For more information, see t

- Using EclipseLink JP
- Cache Type and Ob

Example: `persistence.`

```
<property name="eclipselink.pe
```

Example: property `Map`

```
import org.eclipse.persistence
propertiesMap.put(PersistenceU
```

For information about optimization, see Optimizing the EclipseLink Application

For more information about persistence unit properties, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

Using EclipseLink JPA Extensions for Copy Policy

The copy policy enables EclipseLink to produce exact copies of persistent objects. For more information, see Configuring Copy Policy.

EclipseLink defines the following copy policy annotations:

- `@CopyPolicy`
- `@CloneCopyPolicy`
- `@InstantitationCopyPolicy`

How to Use the @CopyPolicy Annotation

Use the `@CopyPolicy` annotation to specify a class that implements the `org.eclipse.persistence.descriptors.copying.CopyPolicy` interface to set

the copy policy on an `Entity`.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface CopyPolicy {
  Class value();
}
```

You can define the `@CopyPolicy` annotation on the following:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification);
- `@MappedSuperclass`

This table lists attributes of the `@CopyPolicy` annotation.

Attributes of the @CopyPolicy Annotation

Attribute	Description
`value`	Set this attribute to the `Class` of the copy policy that you want to apply to your entity's descriptor. The class must implement `org.eclipse.persistence.descriptors.copying.CopyPolicy`

This example shows how to use the `@CopyPolicy` annotation.

Usage of the @CopyPolicy Annotation

```
@Entity
 @Table(name="EMPLOYEE")
 @CopyPolicy(mypackage.MyCopyPolicy.class)
 public class Employee implements Serializable {
      ...
 }
```

How to Use the @CloneCopyPolicy Annotation

Use the `@CloneCopyPolicy` annotation to set the clone copy policy (`org.eclipse.persistence.descriptors.copying.CloneCopyPolicy`) on an `Entity`.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface CloneCopyPolicy {
  Sting method();
  Sting workingCopyMethod();
```

```
}
```

The `@CloneCopyPolicy` must specify one or both of the `method` or `workingCopyMethod` attributes based on the following:

- Use the `method` for the clone whose function is comparison in conjunction with EclipseLink's `DeferredChangeDetectionPolicy` (see Deferred Change Detection Policy).
- Use the `workingCopyMethod` to clone objects that will be returned, as they are registered in EclipseLink's transactional mechanism (the unit of work).

You can define the `@CloneCopyPolicy` annotation on the following:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification);
- `@MappedSuperclass`

This table lists attributes of the `@CloneCopyPolicy` annotation.

Attributes of the @CloneCopyPolicy Annotation

Attribute	Description	Default	Requi Optio
`method`	Set this attribute to the `String` method name that EclipseLink will use to create a clone to enable comparison by EclipseLink's deferred change detection policy) that you want to apply to your entity's descriptor. Note: you have to set either this attribute, or the `workingCopyMethod`, or both.	no default	option red
`workingCopyMethod`	Set this attribute to the `String` method name that EclipseLink will use to create the object returned when registering an `Object` in an EclipseLink unit of work. Note: you have to set either this attribute, or the `method`, or both.	no default	option red

How to Use the @InstantiationCopyPolicy Annotation

Instantiation copy policy is the default copy policy in EclipseLink. Use the `@InstantiationCopyPolicy` annotation to override other types of copy policies for an `Entity`.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface CloneCopyPolicy {
}
```

You can define the `@InstantiationCopyPolicy` annotation on the following:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification);
- `@MappedSuperclass`

The `@InstantiationCopyPolicy` annotation does not have attributes.

Using EclipseLink JPA Extensions for Declaration of Read-Only Classes

EclipseLink defines one annotation that you can use to declare classes as read-only – `@ReadOnly`.

For more information, see the following:

- Configuring Read-Only Descriptors
- Declaring Read-Only Classes

How to Use the @ReadOnly Annotation

Use the `@ReadOnly` annotation to specify that a class is read-only (see Declaring Read-Only Classes).

Note: Any changes made within a managed instance in a transaction or to a detached instance and merged will have no effect in the context of a read-only entity class.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ReadOnly {}
```

You can define the `@ReadOnly` annotation on the following:

- `@Entity` (see Section 8.1 "Entity" of the JPA Specification);
- `@MappedSuperclass`;
- the root of the inheritance hierarchy (if applicable).

The `@ReadOnly` annotation does not have attributes.

This example shows how to use the `@ReadOnly` annotation.

Usage of the @ReadOnly Annotation

```
@Entity
 @ReadOnly
 public class Employee implements Serializable {
       ...
 }
```

Using EclipseLink JPA Extensions for Returning Policy

The returning policy enables `INSERT` or `UPDATE` operations to return values back into the object being written. These values include table default values, trigger or stored procedures computed values. For more information, see Configuring Returning Policy.

EclipseLink defines the following returning policy annotations:

- `@ReturnInsert`
- `@ReturnUpdate`

How to Use the @ReturnInsert Annotation

You can only specify the `@ReturnInsert` for a `@Basic` mapping.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ReturnInsert {
  boolean returnOnly() default false;
}
```

This table lists attributes of the `@ReturnInsert` annotation.

Attributes of the @ReturnInsert Annotation

Attribute	Description	Default	Required Optional
`returnOnly`	Set this attribute to the `boolean` value of `true` if you want a return of a value for this field, without including the field in the insert.	`false`	optional

The Usage of the @ReturnInsert Annotation Without Arguments example shows how to use the `@ReturnInsert` annotation without specifying the value for the `returnOnly` argument, therefore accepting the default value of `false`. The Usage of the @ReturnInsert Annotation with Arguments example shows how to set the value of the `returnOnly` argument to `true`.

Usage of the @ReturnInsert Annotation Without Arguments

```
@ReturnInsert
 public String getFirstName() {
      return firstName;
 }
```

Usage of the @ReturnInsert Annotation with Arguments

```
@ReturnInsert(returnOnly=true)
 public String getFirstName() {
      return firstName;
 }
```

How to Use the @ReturnUpdate Annotation

You can only specify the `@ReturnUpdate` for a `@Basic` mapping.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ReturnUpdate {}
```

The `@ReturnUpdate` annotation does not have attributes.

This example shows how to use the `@ReturnUpdate` annotation.

Usage of the @ReturnUpdate Annotation

```
@ReturnUpdate
public String getFirstName() {
   return firstName;
}
```

Using EclipseLink JPA Extensions for Optimistic Locking

EclipseLink defines one annotation for optimistic locking – `@OptimisticLocking`.

For more information, see the following:

- Optimistic Locking

- Configuring an Optimistic Locking Policy

How to Use the @OptimisticLocking Annotation

You can use the `@OptimisticLocking` annotation to specify the type of optimistic locking that EclipseLink should use when updating or deleting entities.

Note: EclipseLink supports additional optimistic locking policies beyond what is supported through the JPA specification (such as `@Version` - see Section 9.1.17 "Version Annotation" of the JPA Specification). When mapping to a database schema where a version column does not exist and cannot be added, these locking policies enable the concurrency protection.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface OptimisticLocking {
  OptimisticLockingType type() default VERSION_COLUMN;
  Column[] selectedColumns() default {};
  boolean cascade() default false;
}
```

This table lists attributes of the `@OptimisticLocking` annotation.

Attributes of the @OptimisticLocking Annotation

Attribute	Description

`type`	Set this attribute to the type (`org.eclipse.persistence.annotations.OptimisticLo` enumerated type) of the optimistic locking policy that you will be u The following are the valid values for the `OptimisticLockingT` • `ALL_COLUMNS` – Use this type of locking policy to compare the table in the `WHERE` clause during an update or a delete any field has been changed, EclipseLink will throw an optir exception. • `CHANGED_COLUMNS` – Use this type of locking policy to co fields in the table in the `WHERE` clause during an update op field has been changed, EclipseLink will throw an optimisti exception. Note: performing the same during a delete operation will o primary keys. • `SELECTED_COLUMNS` – Use this type of locking policy to c selected fields in the table in the `WHERE` clause during an u delete operation. If any field has been changed, EclipseLin optimistic locking exception. Note: specified fields must be mapped and must not be pri Note: EclipseLink will throw an exception if you set the `SELECTED_COLUMNS` type, but fail to specify the `selecte` You must also specify the `name` attribute of the `Column`. • `VERSION_COLUMN` – Use this type of locking policy to com version number in the `WHERE` clause during an update ope Note: the version field must be mapped and must not be th Note: this functionality is equivalent to the functionality of th annotation (see Section 9.1.17 "Version Annotation" of the Specification) in JPA. If you use this option, you must also `@Version` annotation on the version field or property. For more information, see What You May Need to Know A Fields.
`selectedColumns`	Set this attribute to an array of `javax.persistence.Column` i For an optimistic locking policy of type SELECTED_COLUMNS, t member becomes a required field. Note: EclipseLink will throw an exception if you set the SELECTE type, but fail to specify the `selectedColumns`. You must also sp attribute of the `Column`.

`cascade`	Set the value of this attribute to a `boolean` value of `true` to spec optimistic locking policy should cascade the lock. By enabling cascading you configure EclipseLink to automatically field update on a parent object when its privately owned child obj field changes. Note: In the current release, only supported with VERSION_COL For more information, see the following: • Optimistic Version Locking Policies and Cascading • Configuring Optimistic Locking Policy Cascading

Note: Setting an `@OptimisticLocking` may override any `@Version` specification (see Section 9.1.17 "Version Annotation" of the JPA Specification) on the entity: EclipseLink will not throw an exception, but will log a warning.

You can specify `@Version` without any `@OptimisticLocking` specification to define a version locking policy (`org.eclipse.persistence.descriptors.VersionLockingPolicy`) on the source entity.

This example shows how to use the `@OptimisticLocking` annotation with the `ALL_COLUMNS` type.

Usage of the @OptimisticLocking Annotation - ALL_COLUMNS

```
@Entity
@Table(name="EMPLOYEE")
@OptimisticLocking(type=OptimisticLockingType.ALL_COLUMNS)
public class Employee implements Serializable{

   private Integer id;
   private String firstName;
   private String lastName;
   ...
}
```

The following example shows how to use the `@OptimisticLocking` annotation with the `CHANGED_COLUMNS` type.

Usage of the @OptimisticLocking Annotation - CHANGED_COLUMNS

```
@Entity
@Table(name="EMPLOYEE")
@OptimisticLocking(type=OptimisticLockingType.CHANGED_COLUMNS)
public class Employee implements Serializable{

   private Integer id;
   private String firstName;
   private String lastName;
```

```
   ...
}
```

The following example shows how to use the `@OptimisticLocking` annotation with the `SELECTED_COLUMNS` type.

Usage of the @OptimisticLocking Annotation - SELECTED_COLUMNS

```
@Entity
@Table(name="EMPLOYEE")
@OptimisticLocking(
   type=OptimisticLockingType.SELECTED_COLUMNS,
   selectedColumns={@Column(name="id"), @Column(name="lastName")}
)
public class Employee implements Serializable{

   @Id
   private Integer id;
   private String lastName;
   private String lastName;
   ...
}
```

The following example shows how to use the `@OptimisticLocking` annotation with the `VERSION_COLUMN` type.

Usage of the @OptimisticLocking Annotation - VERSION_COLUMN

```
@Entity
@Table(name="EMPLOYEE")
@OptimisticLocking(type=OptimisticLockingType.VERSION_COLUMN,
cascade=true)
public class Employee implements Serializable{

   private String firstName;
   private String lastName;
   @Version private int version;
   ...
}
```

Using EclipseLink JPA Extensions for Stored Procedure Query

EclipseLink defines the following stored procedure query annotations:

- `@NamedStoredProcedureQuery`
- `@StoredProcedureParameter`
- `@NamedStoredProcedureQueries`

You can execute a stored procedure query like any other named query (see Named Queries). For more information, see Queries.

How to Use the @NamedStoredProcedureQuery Annotation

Use the `@NamedStoredProcedureQuery` to define queries that call stored procedures as named queries.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedStoredProcedureQuery {
  String name();
  String procedureName();
  QueryHint[] hints() default {};
  Class resultClass() default void.class;
  String resultSetMapping() default "";
  boolean returnsResultSet() default false;
  StoredProcedureParameter[] parameters() default {};
}
```

This table lists attributes of the `@NamedStoredProcedureQuery` annotation.

Attributes of the @NamedStoredProcedureQuery Annotation

Attribute	Description	Default
`name`	Set this attribute to the unique `String` name that references this stored procedure query.	no default
`hints`	Set this attribute to an array of `javax.persistence.QueryHint` instances.	empty `Qu`
`resultClass`	Set this attribute to the `Class` of the query result.	`void.cl`
`resultSetMapping`	Set this attribute to the `String` name of the `javax.persistence.SQLResultSetMapping` instance.	empty `St`
`procedureName`	Set this attribute to the `String` name of the stored procedure.	no default
`returnsResultSet`	Set this attribute to the `boolean` value of `false` to disable the return of the result set.	`true`
`parameters`	Set the value of this attribute to an array of `@StoredProcedureParameter` instances to define arguments to the stored procedure.	empty `StoredP` array

This example shows how to use the `@NamedStoredProcedureQuery` annotation.

Usage of the @NamedStoredProcedureQuery Annotation

```
@Entity
@Table(name="EMPLOYEE")
@NamedStoredProcedureQuery(
   name="ReadEmployee",
   procedureName="Read_Employee",
   parameters={
      @StoredProcedureParameter(queryParamater="EMP_ID")}
)
public class Employee implements Serializable{
   ...
}
```

How to Use the @StoredProcedureParameter Annotation

Use the `@StoredProcedureParameter` annotation within a @NamedStoredProcedureQuery annotation to define arguments to the stored procedure.

```
@Target({})
@Retention(RUNTIME)
public @interface StoredProcedureParameter {
  String queryParameter();
  Direction direction() default IN;
  int jdbcType() default -1;
  String jdbcTypeName() default "";
  String name() default "";
  Class type() default void.class;
}
```

This table lists attributes of the `@StoredProcedureParameter` annotation.

Attributes of the @StoredProcedureParameter Annotation

Attribute	Description	D
`queryParameter`	Set this attribute to the `String` query parameter name.	n
`direction`	Set the value of this attribute to define the direction(`org.eclipse.persistence.annotations.Direction` enumerated type) of the stored procedure parameter. The following are valid values for `Direction.IN`: • `IN` – Input parameter. • `OUT` – Output parameter.	e D

	• `IN_OUT` – Input and output parameter. • `OUT_CURSOR` – Output cursor. Note: EclipseLink will throw an exception if you set more than one parameter to the `OUT_CURSOR` type.	
`name`	Set this attribute to the `String` name of the stored procedure parameter.	"
`type`	Set this attribute to the type of Java `Class` that you want to receive back from the procedure. This depends on the type returned from the procedure.	v
`jdbcType`	Set this attribute to the int value of JDBC type code. This depends on the type returned from the procedure.	-1
`jdbcTypeName`	Set this attribute to the `String` value of the JDBC type name. Note: setting of this attribute may be required for `ARRAY` or `STRUCT` types.	"

The Usage of the @NamedStoredProcedureQuery Annotation example shows how to use the `@StoredProcedureParameter` annotation.

For more information, see the following:

- StoredProcedureCall
- Using a StoredProcedureCall
- Call Queries

How to Use the @NamedStoredProcedureQueries Annotation

Use the `@NamedStoredProcedureQueries` to define queries that call stored procedures as named queries.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface NamedStoredProcedureQueries {
  NamedStoredProcedureQuery[] value();
}
```

This table lists attributes of the `@NamedStoredProcedureQueries` annotation.

Attributes of the @NamedStoredProcedureQueries Annotation

Attribute	Description	Default	Require Optiona

`value`	Set this attribute to the array of the `@NamedStoredProcedureQuery` annotations.	no default	required

Using EclipseLink JPA Extensions for JDBC

EclipseLink JPA provides persistence unit properties that you can define in a `persistence.xml` file to configure how EclipseLink will use the connections returned from the data source used. These properties are divided into the two following categories:

- Options that you can use to configure how EclipseLink communicates with the JDBC connection (see How to Use EclipseLink JPA Extensions for JDBC Connection Communication).
- Options that you can use to configure EclipseLink own connection pooling (see How to Use EclipseLink JPA Extensions for JDBC Connection Pooling).

How to Use EclipseLink JPA Extensions for JDBC Connection Communication

This table lists the EclipseLink JPA persistence unit properties that you can define in a `persistence.xml` file to configure how EclipseLink communicates with the JDBC connection.

EclipseLink JPA Persistence Unit Properties for JDBC Connection Communication

Property	Usage

`eclipselink.jdbc.bind-parameters`	Control whether or not the query uses paramete
	
	For more information, see How to Use Paramete Prepared Statement Caching for Optimization.
	
	The following are the valid values:
	
	• `true` – bind all parameters.
	• `false` – do not bind parameters.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.bind-parameters" val`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_BIND`

`eclipselink.jdbc.native-sql`	Enable or disable EclipseLink's generation of da to generic SQL).[2]
	
	The following are the valid values:
	
	• `true` – enable EclipseLink's generation o
	• `false` – disable generation of database p
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.native-sql" value="t`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.NATIVE_SQ`

`eclipselink.jdbc.batch-writing`	Specify the use of batch writing to optimize trans Set the value of this property into the session at The following are the valid values for the use in a `org.eclipse.persistence.config.Batch` • `JDBC` – use JDBC batch writing. • `Buffered` – do not use either JDBC batc • `Oracle-JDBC` – use both JDBC batch wr writing. Use `OracleJDBC` in your property map. • `None` – do not use batch writing (turn it of

	Note: if you set any other value, EclipseLink will
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.batch-writing" value`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.BATCH_WRI`

`eclipselink.jdbc.cache-statements`	Enable or disable EclipseLink internal statement
	
	Note: we recommend enabling this functionality pooling.
	
	The following are the valid values:
	
	• `true` – enable EclipseLink's internal state
	• `false` – disable internal statement cachi
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.cache-statements" va`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.CACHE_STA`

`eclipselink.jdbc.cache-statements.size`	The number of statements held when using inter
	
	Set the value at the deployment time.
	
	Valid values: 0 to `Integer.MAX_VALUE` (deper
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.cache-statements.siz`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.CACHE_STA`

`eclipselink.jdbc.exclusive-connection.is-lazy`	Specify when a write connection is acquired lazi
	Connection Acquisition.
	
	The following are the valid values:
	
	• `true` - aquire the write connection lazily.
	• `false` - do not aquire the write connectio
	
	For more information, see Configuring Connecti
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.exclusive-connection`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.EXCLUSIVE`

`eclipselink.jdbc.exclusive-connection.mode`	Specify when EclipseLink should perform reads information, see Exclusive Write Connections. You can set this property while creating either a passed to the `createEntityManagerFactor` file), or an `EntityManager` (in the map passed Note that the latter overrides the former. The following are the valid values for the use in `org.eclipse.persistence.config.Exclu` • `Transactional` - Create an isolated cli some or all entities require isolated cache Note: EclipseLink keeps the connection e Inside the transaction, EclipseLink perfor connection. However, outside the Eclipeli from the connection pool for each read a is executed. • `Isolated` - Create an exclusive isolated `Entity`; otherwise, raise an error. Note: EclipseLink keeps the connection e `EntityManager`. Inside the transaction, through the exclusive connection. Howev isolated entities are read through the excl new connection is acquired from the conn immediately after the query is executed. • `Always` - Create an exclusive isolated cli reading an isolated `Entity`; otherwise, c Note: EclipseLink keeps the connection e `EntityManager` and performs all writes

	For more information, see Configuring Connecti
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.exclusive-connection`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.EXCLUSIVE`

`eclipselink.jdbc.driver`	The class name of the JDBC driver you want to
	This class must be on your application classpath
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.driver" value="oracl`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_DRIV`

`eclipselink.jdbc.password`	The password for your JDBC user.[3]
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.password" value="tig`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_PASS`

`eclipselink.jdbc.url`	The JDBC connection URL required by your JDB
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.url" value="jdbc:ora`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_URL,`

`eclipselink.jdbc.user`	The user name for your JDBC user.[3]

	Example: `persistence.xml` file
	`<property name="eclipselink.jdbc.url" value="scott"/>`
	Example: property `Map`
	`import org.eclipse.persistence.config.PersistenceUnit`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_USER`

[1] This property applies when used in a Java SE environment.

[2] This property applies when used both in a Java SE and Java EE environment.

[3] This property applies when used in a Java SE environment or a resource-local persistence unit (see Section 5.5.2 "Resource-Local Entity Managers" and Section 6.2.1.2 "transaction-type" of the JPA Specification).

[4] To do this, set the `eclipselink.cache.shared.<ENTITY>` property for one or more entities to `false`; Use the `eclipselink.cache.shared.default` property if you want to use the isolated cache for all entities.

For information about the use of annotations as opposed to persistence unit properties and vice versa, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

For more information about persistence unit properties, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

How to Use EclipseLink JPA Extensions for JDBC Connection Pooling

This table lists the EclipseLink JPA persistence unit properties that you can define in a `persistence.xml` file to configure EclipseLink internal connection pooling.

EclipseLink JPA Persistence Unit Properties for JDBC Connection Pooling

Property	Usage

`eclipselink.jdbc.read-connections.max`	The maximum number of connections allowed in the J pool.[1]
	
	Valid values: 0 to `Integer.MAX_VALUE` (depending `String`.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.read-connections.max" valu`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnitProper`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_READ_CONNE`

`eclipselink.jdbc.read-connections.min`	The minimum number of connections allowed in the JI
	
	Valid values: 0 to `Integer.MAX_VALUE` (depending `String`.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.read-connections.min" valu`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnitProper`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_READ_CONNE`

`eclipselink.jdbc.read-connections.shared`	Specify whether or not to allow concurrent use of shar
	
	The following are the valid values:
	
	• `true` – allow concurrent use of shared read co
	• `false` – do not allow the concurrent use of sha concurrent readers are each allocated their own
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.read-connections.shared" v`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnitProper`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_READ_CONNE`

`eclipselink.jdbc.write-connections.max`	The maximum number of connections allowed in the J pool.[1] Valid values: to `Integer.MAX_VALUE` (depending on

	`String`.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.write-connections.max" val`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnitProper`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_WRITE_CONN`

`eclipselink.jdbc.write-connections.min`	The maximum number of connections allowed in the J pool.[1]
	
	Valid values: 0 to `Integer.MAX_VALUE` (depending `String`.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.jdbc.write-connections.min" val`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnitProper`
	`propertiesMap.put(PersistenceUnitProperties.JDBC_WRITE_CONN`

[1] This property applies when used in a Java SE environment.

For information about the use of annotations as opposed to persistence unit properties and vice versa, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

For more information about persistence unit properties, see hat You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

Using EclipseLink JPA Extensions for Logging

This table lists the EclipseLink JPA persistence unit properties that you can define in a `persistence.xml` file to configure EclipseLink logging. Additional information (including examples) is available in How to configure a custom logger in JPA.

EclipseLink JPA Persistence Unit Properties for Logging

Property	Usage

`eclipselink.logging.logger`	Select the type of logger to use.
	
	The following are the valid values for the use `org.eclipse.persistence.config.Pe`
	
	• `DefaultLogger` – the EclipseLink n `org.eclipse.persistence.logg`
	• `JavaLogger` – the `java.util.log` `org.eclipse.persistence.logg`
	• `ServerLogger` – the `java.util.l` `org.eclipse.persistence.plat` the application server's logging as de `org.eclipse.persistence.plat`
	• Fully qualified class name of a custon the `org.eclipse.persistence.l`
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.logging.logger" valu`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persistenc`
	`propertiesMap.put(PersistenceUnitProperties.LOGG`

`eclipselink.logging.level`	Control the amount and detail of log output b of information). The following are the valid values for the use `java.util.logging.Level`: • `OFF` – disables logging. You may want to set logging to `OFF` c logging. • `SEVERE` – logs exceptions indicating exceptions generated during login. Th • `WARNING` – logs exceptions that do n exceptions not logged with severe lev • `INFO` – logs the login/logout per seve acquiring the session, detailed inform • `CONFIG` – logs only login, JDBC conn You may want to use the `CONFIG` log • `FINE` – logs SQL. You may want to use this log level du production time. • `FINER` – similar to `WARNING`. Include You may want to use this log level du production time. • `FINEST` – includes additional low leve

	You may want to use this log level du
	production time.
	• `ALL` – logs at the same level as `FINE`
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.logging.level" value`
	
	Example: property `Map`
	
	`import java.util.logging.Level;`
	`import org.eclipse.persistence.config.Persistenc`
	`propertiesMap.put(PersistenceUnitProperties.LOGG`

`eclipselink.logging.timestamp`	Control whether the timestamp is logged in
	
	The following are the valid values:
	
	• `true` – log a timestamp.
	• `false` – do not log a timestamp.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.logging.timestamp" v`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persistenc`
	`propertiesMap.put(PersistenceUnitProperties.LOGG`

`eclipselink.logging.thread`	Control whether a thread identifier is logged
	
	The following are the valid values:
	
	• `true` – log a thread identifier.
	• `false` – do not log a thread identifier
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.logging.thread" valu`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persistenc`
	`propertiesMap.put(PersistenceUnitProperties.LOGG`

`eclipselink.logging.session`	Control whether an EclipseLink session ider
	
	The following are the valid values:
	
	• `true` – log an EclipseLink session id
	• `false` – do not log an EclipseLink se
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.logging.session" val`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persistenc`
	`propertiesMap.put(PersistenceUnitProperties.LOGG`

`eclipselink.logging.exceptions`	Control whether the exceptions thrown from
	returning the exception to the calling applica
	and not masked by the application code.
	
	The following are the valid values:
	
	• `true` – log all exceptions.
	• `false` – do not log exceptions.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.logging.exceptions"`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persistenc`
	`propertiesMap.put(PersistenceUnitProperties.LOGG`

`eclipselink.logging.file`	Specify a file location for the log output (inst
	
	Valid values: a string location to a directory
	may be relative to your current working direc
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.logging.file" value=`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Persistenc`
	`propertiesMap.put(PersistenceUnitProperties.LOGG`

[1] This property applies when used in a Java SE environment.

For information about the use of annotations as opposed to persistence unit properties

and vice versa, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

For more information about persistence unit properties, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

Using EclipseLink JPA Extensions for Session, Target Database and Target Application Server

This table lists the EclipseLink JPA persistence unit properties that you can define in a `persistence.xml` file to configure EclipseLink extensions for session, as well as the target database and application server.

EclipseLink JPA Persistence Unit Properties for Database, Session, and Application Server

Property	Usage

`eclipselink.session-name`	Specify the name by option if you need to use a pre-existing Ec
	
	Valid values: a valid
	
	Example: `persiste`
	
	`<property name="eclipse`
	
	Example: property `M`
	
	`import org.eclipse.pers`
	`propertiesMap.put(Persi`

`eclipselink.sessions-xml`	Specify persistence i (`sessions.xml`). You can use this opti property, EclipseLink `persistence.xml`, see hat You May Nee Indicate the session Note: If you do not sp Valid values: the reso

	Example: persiste <property name="eclipse **Example**: property M import org.eclipse.pers propertiesMap.put(Persi
eclipselink.session-event-listener	Specify a descriptor e For more information Valid values: qualifie org.eclipse.pers **Example**: persiste <property name="eclipse **Example**: property M import org.eclipse.pers propertiesMap.put(Persi
eclipselink.session.include.descriptor.queries	Enable or disable the These queries includ so on. The following are the • true – enable • false – disab **Example**: persiste <property name="eclipse **Example**: property M import org.eclipse.pers propertiesMap.put(Persi
eclipselink.target-database	Specify the type of da The following are the org.eclipse.pers • Attunity – c • Auto – Eclipse determine the

	• Cloudscape – • Database – c database is no the Auto optio • DB2 – configur • DB2Mainfram • DBase – confi • Derby – confi • HSQL – configu • Informix – c • JavaDB – con • MySQL – confi • Oracle – con • PointBase – • PostgreSQL – • SQLAnywhere • SQLServer – • Sybase – con • TimesTen – c You can also set the org.eclipse.pers **Example**: persiste <property name="eclipse **Example**: property Ma import org.eclipse.pers import org.eclipse.pers propertiesMap.put(Persi
eclipselink.target-server	Specify the type of ap The following are the org.eclipse.pers • None – configu • WebLogic – c Note: this serv disabled. • WebLogic_9 – • WebLogic_10 10. • OC4J – configu • SunAS9 – con Note: this serv disabled. • WebSphere –

	• WebSphere_6 version 6.1. • JBoss – confi • Fully qualified org.eclipse **Example**: persiste <property name="eclipse **Example**: property Ma import org.eclipse.pers import org.eclipse.pers propertiesMap.put(Persi

For information about the configuration of platforms, see the following:

- Target Platforms
- Integrating EclipseLink with an Application Server
- Projects and Platforms
- Database Platforms

For information about the use of annotations as opposed to persistence unit properties and vice versa, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

For more information about persistence unit properties, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

Using EclipseLink JPA Extensions for Schema Generation

This table lists the EclipseLink JPA persistence unit properties that you can define in a `persistence.xml` file to configure schema generation.

EclipseLink JPA Persistence Unit Properties for Schema Generation

Property	Usage
eclipselink.ddl-generation	Specify what Data Definition Language (DDL) genera generation target, see eclipselink.ddl-genera The following are the valid values for the use in a pe • none – EclipseLink does not generate DDL; n • create-tables – EclipseLink will attempt to already exists, EclipseLink will follow the defa

	combination (when a `CREATE TABLE` SQL is exception is thrown and the table *is not* create also eclipselink.create-ddl-jdbc-fi
	• `drop-and-create-tables` – EclipseLink w issues are encountered, EclipseLink will follow driver combination, then continue with the nex file-name and eclipselink.drop-ddl-
	
	The following are the valid values for the `org.eclipse.persistence.config.Persiste`
	
	• `NONE` – see `none`.
	• `CREATE_ONLY` – see `create-tables`.
	• `DROP_AND_CREATE` – see `drop-and-creat`
	
	If you are using persistence in a Java SE environme tables, additionally define a Java system property `IN`
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.ddl-generation" value="create-`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnitPrope`
	`propertiesMap.put(PersistenceUnitProperties.DDL_GENERATION`

`eclipselink.application-location`	Specify where EclipseLink should write generated DI and eclipselink.drop-ddl-jdbc-file-name) to anything other than `none`.
	
	Valid values: a file specification to a directory in whic to your current working directory or absolute. If it doe for your operating system.
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.application-location" value="C`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.PersistenceUnitPrope`
	`propertiesMap.put(PersistenceUnitProperties.APP_LOCATION,`

`eclipselink.create-ddl-jdbc-file-name`	Specify the file name of the DDL file that EclipseLink entities. This file is written to the location specified by eclipselink.ddl-generation is set to `create` Valid values: a file name valid for your operating sys long as the concatenation of eclipselink.appli

	file-name is a valid file specification for your opera
	
	Example: persistence.xml file
	
	`<property name="eclipselink.create-ddl-jdbc-file-name" val`
	
	Example: property Map
	
	`import org.eclipse.persistence.config.PersistenceUnitPrope`
	`propertiesMap.put(PersistenceUnitProperties.CREATE_JDBC_DD`

eclipselink.drop-ddl-jdbc-file-name	Specify the file name of the DDL file that EclipseLink JPA entities. This file is written to the location specifi eclipselink.ddl-generation is set to drop-a
	
	Valid values: a file name valid for your operating sys long as the concatenation of eclipselink.appli file-name is a valid file specification for your opera
	
	Example: persistence.xml file
	
	`<property name="eclipselink.drop-ddl-jdbc-file-name" value`
	
	Example: property Map
	
	`import org.eclipse.persistence.config.PersistenceUnitPrope`
	`propertiesMap.put(PersistenceUnitProperties.DROP_JDBC_DDL_`

eclipselink.ddl-generation.output-mode	Use this property to specify the DDL generation targ The following are the valid values for the use in the p • both – generate SQL files and execute them If eclipselink.ddl-generation is set to file-name is written to eclipselink.app If eclipselink.ddl-generation is set to eclipselink.create-ddl-jdbc-file-n written to eclipselink.application-loc • database – execute SQL on the database or • sql-script – generate SQL files only (do n If eclipselink.ddl-generation is set to file-name is written to eclipselink.app If eclipselink.ddl-generation is set to eclipselink.create-ddl-jdbc-file-n written to eclipselink.application-loc The following are the valid values for the org.eclipse.persistence.config.Persiste • DDL_BOTH_GENERATION – see both.

	• DDL_DATABASE_GENERATION – see databa
	• DDL_SQL_SCRIPT_GENERATION – see sql-
	
	Example: persistence.xml file
	
	`<property name="eclipselink.ddl-generation.output-mode" va`
	
	Example: property Map
	
	`import org.eclipse.persistence.config.PersistenceUnitPrope`
	`propertiesMap.put(PersistenceUnitProperties.DDL_GENERATION`

For information about the use of annotations as opposed to persistence unit properties and vice versa, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

For more information about persistence unit properties, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

Using EclipseLink JPA Extensions for Tracking Changes

Within a transaction, EclipseLink automatically tracks entity changes.

EclipseLink defines one annotation for tracking changes – `@ChangeTracking` annotation.

EclipseLink also provides a number of persistence unit properties that you can specify to configure EclipseLink change tracking (see How to Use the Persistence Unit Properties for Change Tracking). These properties may compliment or provide an alternative to the usage of annotations.

Note: Persistence unit properties always override the corresponding annotations' attributes. For more information, see What You May Need to Know About Overriding Annotations in JPA and What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

For more information, see Unit of Work and Change Policy.

How to Use the @ChangeTracking Annotation

Use the `@ChangeTracking` annotation to set the unit of work's change policy.

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ChangeTracking {
  ChangeTrackingType value() default AUTO;
}
```

Note: This is an optimization feature that lets you tune the way EclipseLink detects changes. You should choose the strategy based on the usage and data modification patterns of the entity type as different types may have different access patterns and hence different settings, and so on.

For more information, see the following:

- Optimizing the Unit of Work
- Read Optimization Examples
- Write Optimization Examples

This table lists attributes of the `@ChangeTracking` annotation.

Attributes of the @ChangeTracking Annotation

Attribute	Description
`value`	• `OBJECT` – This option uses weaving to detect if the object has been changed, but uses a backup copy of the object to determine what fields or properties changed. This is more efficient than the `DEFERRED` option, but less efficient than the `ATTRIBUTE` option. You must enable weaving to use this option. This option supports additional mapping configuration to attribute. For more information, see Object-Level Change Tracking Policy. • `DEFERRED` – This option defers all change detection to the `UnitOfWork`'s change detection process. This option uses a backup copy of the object to determine which fields or properties changed. This is the least efficient option, but does not require weaving and supports additional mapping configurations to attribute and object. For more information, see Deferred Change Detection Policy. • `AUTO` – This option does not set any change tracking policy. The policy is determined by the EclipseLink agent: if the class can be weaved for change tracking the `ATTRIBUTE` option is used; otherwise, the `DEFERRED` option is used. `AUTO` is the only change tracking value where EclipseLink chooses the value; any other value will explicitly cause EclipseLink to use that value, so if you decide to explicitly set it, you must set it correctly and

	use your model correctly to ensure the change tracking detects your changes. Note: For every option, objects with changed attributes will be processed at the commit time to include any changes in the results of the commit. Unchanged objects will be ignored. Note: The weaving of change tracking is the same for attribute and object change tracking. For information about the relationship between the `value` attribute and `eclipselink.weaving.changetracking` property, see What You May Need to Know About the Relationship Between the Change Tracking Annotation and Persistence Unit Property

This example shows how to use the `@ChangeTracking` annotation.

Usage of @ChangeTracking Annotation

```
@Entity
@Table(name="EMPLOYEE")
@ChangeTracking(OBJECT) (
public class Employee implements Serializable {
    ...
}
```

Note: You cannot use the attribute change tracking in conjunction with the following:

- optimistic field-level locking (see Optimistic Locking);
- mutable types (such as mutable temporals or mutable serialized mappings);
- non-lazy collection mappings.

The attribute change tracking will not detect object changes made through reflection or direct field access to fields mapped as properties.

How to Use the Persistence Unit Properties for Change Tracking

This table lists the persistence unit properties that you can define in a `persistence.xml` file to configure EclipseLink change tracking.

EclipseLink JPA Persistence Unit Properties for Change Tracking

Property	Usage

`eclipselink.weaving.changetracking`	Enable or disable the `AttributeLev`
	
	The following are the valid values:
	
	• `true` – enable the `Attribute` weaving. When enabled, only c change tracking have change tr
	• `false` – disable the `Attribut` weaving. Use this setting if the
	  • you cannot weave at all;
	  • you do not want your cla example, for debugging
	  • you wish to disable this f support it (for example, y or `java.util.Calenda` modifying the underlying
	
	Note: you may set this option only if th set to `true`. The purpose of the `eclipselink.weaving.changetr` granular control over weaving.
	
	For more information, see the followin
	
	• Using EclipseLink JPA Weaving
	• Configuring Change Policy
	
	Example: `persistence.xml` file
	
	`<property name="eclipselink.weaving.change`
	
	Example: property `Map`
	
	`import org.eclipse.persistence.config.Pers`
	`propertiesMap.put(PersistenceUnitPropertie`

For information about the relationship between the `value` attribute of the `@ChangeTracking` annotation and `eclipselink.weaving.changetracking` property, see What You May Need to Know About the Relationship Between the Change Tracking Annotation and Persistence Unit Property.

For information about the use of annotations as opposed to persistence unit properties and vice versa, see the following:

- What You May Need to Know About Overriding Annotations in JPA
- What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

For more information about persistence unit properties, see What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties.

What You May Need to Know About the Relationship Between the Change Tracking Annotation and Persistence Unit Property

The following table shows the dependency between the `value` attribute of the `@ChangeTracking` annotation and the `eclipselink.weaving.changetracking` property (see How to Use the Persistence Unit Properties for Change Tracking).

Relationship Between the Change Tracking Annotation and Property

@ChangeTracking(value=)	eclipselink.weaving.changetracking=	Description
`ATTRIBUTE`	`true`	Weave and enable attribu Even if the descriptors co by change tracking weavi use your set methods to e raise the events for these
`ATTRIBUTE`	`false`	Do not weave change tra You must implement the `org.eclipse.persist` interface and raise the ch descriptor initialization.
`OBJECT`	`true`	Object change tracking is Note: Weaving is identica
`OBJECT`	`false`	Do not weave change tra You must implement the `org.eclipse.persist` interface and raise the ch descriptor initialization.
`DEFERRED`	`true`	Do not weave change tra Deferred change tracking
`DEFERRED`	`false`	Do not weave change tra Deferred change tracking
`AUTO`	`true`	Weave change tracking if that do not support chang

AUTO	false	Do not weave change tra Deferred change tracking

Using EclipseLink JPA Query Customization Extensions

This section describes the following:

- How to Use EclipseLink JPA Query Hints
- How to Use EclipseLink Query API in JPA Queries

How to Use EclipseLink JPA Query Hints

The EclipseLink JPA Query Hints table lists the EclipseLink JPA query hints that you can specify when you construct a JPA query, as the Specifying an EclipseLink JPA Query Hint example shows, or when you specify a JPA query using the `@QueryHint` annotation (see Section 8.3.1 "NamedQuery Annotation" of the JPA Specification), as the Specifying an EclipseLink JPA Query Hint with @QueryHint example shows.

All EclipseLink query hints are defined in the `QueryHints` class in the `org.eclipse.persistence.config` package.

The EclipseLink query hints include the following:

- `eclipselink.cache-usage`
- `eclipselink.query-type`
- `eclipselink.jdbc.bind-parameters`
- `eclipselink.pessimistic-lock`
- `eclipselink.refresh`
- `eclipselink.refresh.cascade`
- `eclipselink.batch`
- `eclipselink.join-fetch`
- `eclipselink.read-only`
- `eclipselink.jdbc.timeout`
- `eclipselink.jdbc.fetch-size`
- `eclipselink.jdbc.max-rows`
- `eclipselink.result-collection-type`

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`

The Specifying an EclipseLink JPA Query Hint and Specifying an EclipseLink JPA Query

Hint with @QueryHint examples show how to set the value of hint `eclipselink.jdbc.bind-parameters` using the `QueryHints` configuration class to set the name of the hint, and the `HintValues` configuration class to set the value.

Specifying an EclipseLink JPA Query Hint

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;

Customer customer =
(Customer)entityMgr.createNamedQuery("findCustomerBySSN").
   setParameter("SSN", "123-12-1234").
   setHint(QueryHints.BIND_PARAMETERS,
HintValues.PERSISTENCE_UNIT_DEFAULT).
   getSingleResult();
```

Specifying an EclipseLink JPA Query Hint with @QueryHint

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;

@Entity
@NamedQuery(
   name="findEmployeeByDept",
   query="SELECT e FROM Employee e WHERE e.dept=:deptNum",
   hints=@QueryHint(name=QueryHints.BIND_PARAMETERS,
value=HintValues.TRUE)
)
public class Employee implements Serializable {
   ...
}
```

Cache Usage

The `eclipselink.cache-usage` hint specifies how the query should interact with the EclipseLink cache.

EclipseLink JPA uses a shared cache mechanism that is scoped to the entire persistence unit. When operations are completed in a particular persistence context, the results are merged back into the shared cache so that other persistence contexts can use them. This happens regardless of whether the entity manager and persistence context are created in Java SE or Java EE. Any entity persisted or removed using the entity manager will always be kept consistent with the cache.

For more information, see the following:

- Session Cache
- How to Use In-Memory Queries

The following are the valid values for the `org.eclipse.persistence.config.CacheUsage`:

- `DoNotCheckCache` – Always go to the database.
- `CheckCacheByExactPrimaryKey` – If a read-object query contains an expression where the primary key is the only comparison, you can obtain a cache hit if you process the expression against the object in memory

- `CheckCacheByPrimaryKey` – If a read-object query contains an expression that compares at least the primary key, you can obtain a cache hit if you process the expression against the objects in memory.
- `CheckCacheThenDatabase` – You can configure any read-object query to check the cache completely before you resort to accessing the database.
- `CheckCacheOnly` – You can configure any read-all query to check only the parent session cache (shared cache) and return the result from it without accessing the database.
- `ConformResultsInUnitOfWork` – You can configure any read-object or read-all query within the context of a unit of work to conform the results with the changes to the object made within that unit of work. This includes new objects, deleted objects and changed objects.
  For more information, see Using Conforming Queries and Descriptors.
- `UseEntityDefault` – Use the cache configuration as specified by the EclipseLink descriptor API for this entity.
  Note: the entity default value is to not check the cache (`DoNotCheckCache`). The query will access the database and synchronize with the cache. Unless refresh has been set on the query, the cached objects will be returned without being refreshed from the database. EclipseLink does not support the cache usage for native queries or queries that have complex result sets such as returning data or multiple objects.

Default: `CacheUsage.UseEntityDefault`
Note: this default is `DoNotCheckCache`.

For more information, see Configuring Cache Usage for In-Memory Queries.

Example: JPA Query API

```
import org.eclipse.persistence.config.CacheUsage;
import org.eclipse.persistence.config.QueryHints;
query.setHint(QueryHints.CACHE_USAGE, CacheUsage.CheckCacheOnly);
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.CacheUsage;
import org.eclipse.persistence.config.TargetDatabase;
@QueryHint(name=QueryHints.CACHE_USAGE,
value=CacheUsage.CheckCacheOnly);
```

Query Type

The `eclipselink.query-type` hint specifies the EclipseLink query type to use for the query.

For most JP QL queries, the `org.eclipse.persistence.queries.ReportQuery` or `org.eclipse.persistence.queries.ReadAllQuery` are used. The `eclipselink.query-type` hint lets you use other query types, such as `org.eclipse.persistence.queries.ReadObjectQuery` for queries that are know to return a single object.

For more information, see the following:

- Object-Level Read Query

- Data-Level Modify Query

The following are the valid values for the `org.eclipse.persistence.config.QueryType`:

- `Auto` – EclipseLink chooses the type of query to use.
- `ReadAll` – Use the ReadAllQuery type for the query.
- `ReadObject` – Use the ReadObjectQuery type for the query.
- `Report` – Use the Report Query type for the query.

Default: `QueryType.Auto`

Example: JPA Query API

```
import org.eclipse.persistence.config.QueryType;
import org.eclipse.persistence.config.QueryHints;
query.setHint(QueryHints.QUERY_TYPE, QueryType.ReadObject);
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.QueryType;
import org.eclipse.persistence.config.TargetDatabase;
@QueryHint(name=QueryHints.QUERY_TYPE,
value=QueryType.ReadObject);
```

Bind Parameters

The `eclipselink.jdbc.bind-parameters` hint controls whether or not the query uses parameter binding.

For more information, see How to Use Parameterized SQL (Parameter Binding) and Prepared Statement Caching for Optimization.

The following are the valid values for the `org.eclipse.persistence.config.HintValues`:

- `TRUE` – bind all parameters.
- `FALSE` – do not bind all parameters.
- `PERSISTENCE_UNIT_DEFAULT` – use the parameter binding setting made in your EclipseLink session's database login, which is `true` by default.

  For more information, see Configuring JDBC Options.

Default: `HintValues.PERSISTENCE_UNIT_DEFAULT`

Example: JPA Query API

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
query.setHint(QueryHints.BIND_PARAMETERS, HintValues.TRUE);
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.HintValues;
```

```
import org.eclipse.persistence.config.TargetDatabase;
@QueryHint(name=QueryHints.BIND_PARAMETERS,
value=HintValues.TRUE);
```

Fetch Size

The `eclipselink.jdbc.fetch-size` hint specifies the number of rows that should be fetched from the database when more rows are needed[1]

For large queries that return a large number of objects you can configure the row fetch size used in the query to improve performance by reducing the number database hits required to satisfy the selection criteria. Most JDBC drivers default to a fetch size of 10, so if you are reading 1000 objects, increasing the fetch size to 256 can significantly reduce the time required to fetch the query's results. The optimal fetch size is not always obvious. Usually, a fetch size of one half or one quarter of the total expected result size is optimal. Note that if you are unsure of the result set size, incorrectly setting a fetch size too large or too small can decrease performance.

A value of 0 means the JDBC driver default will be used.

Valid values: 0 to `Integer.MAX_VALUE` (depending on your JDBC driver) as a `String`.

Default: 0

Note: this value indicates that the JDBC driver default will be used.

[1] This property is dependent on the JDBC driver support.

Timeout

The `eclipselink.jdbc.timeout` hint specifies the number of seconds EclipseLink will wait on a query before throwing a `DatabaseException`[1]

A value of 0 means EclipseLink will never time-out a query.

Valid values: 0 to `Integer.MAX_VALUE` (depending on your JDBC driver) as a `String`.

Default: 0

[1] This property is dependent on the JDBC driver support.

Pessimistic Lock

The `eclipselink.pessimistic-lock` hint controls whether or not pessimistic locking is used.

The following are the valid values for the `org.eclipse.persistence.config.PessimisticLock`:

- `NoLock` – pessimistic locking is not used.

- `Lock` – EclipseLink issues a `SELECT .... FOR UPDATE`.
- `LockNoWait` – EclipseLink issues a `SELECT .... FOR UPDATE NO WAIT`.

Default: `NoLock`

Example: JPA Query API

```
import org.eclipse.persistence.config.PessimisticLock;
import org.eclipse.persistence.config.QueryHints;
query.setHint(QueryHints.PESSIMISTIC_LOCK,
PessimisticLock.LockNoWait);
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.PessimisticLock;
import org.eclipse.persistence.config.QueryHints;
@QueryHint(name=QueryHints.PESSIMISTIC_LOCK,
value=PessimisticLock.LockNoWait);
```

Batch

The `eclipselink.batch` hint supplies EclipseLink with batching information so subsequent queries of related objects can be optimized in batches instead of being retrieved one-by-one or in one large joined read. Batch reading is more efficient than joining because it avoids reading duplicate data.

Batching is only allowed on queries that have a single object in their select clause.

Valid values: a single-valued relationship path expression.

Note: use dot notation to access nested attributes. For example, to batch-read an employee's manager's address, specify `e.manager.address`

Example: JPA Query API

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
query.setHint("eclipselink.batch", "e.address");
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
@QueryHint(name=QueryHints.BATCH, value="e.address");
```

Join Fetch

The `eclipselink.join-fetch` hint allows joining of the attributes.

This is similar to `eclipselink.batch`: subsequent queries of related objects can be optimized in batches instead of being retrieved in one large joined read.

This is different from JP QL joining because it allows multilevel fetch joins.

For more information, see Section 4.4.5.3 "Fetch Joins" of JPA specification.

Valid values: a relationship path expression.

Note: use dot notation to access nested attributes.

Example: JPA Query API

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
query.setHint("eclipselink.join-fetch", "e.address");
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
@QueryHint(name=QueryHints.FETCH, value="e.address");
```

Refresh

The `eclipselink.refresh` hint controls whether or not to update the EclipseLink session cache with objects that the query returns.

The following are the valid values for the `org.eclipse.persistence.config.HintValues`:

- `TRUE` – refresh cache.
- `FALSE` – do not refresh cache.

Default: `FALSE`

Example: JPA Query API

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
query.setHint(QueryHints.REFRESH, HintValues.TRUE);
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
@QueryHint(name=QueryHints.REFRESH, value=HintValues.TRUE);
```

Read Only

The `eclipselink.read-only` hint retrieves read-only results back from the query: on nontransactional read operations, where the requested entity types are stored in the shared cache, you can request that the shared instance be returned instead of a detached copy.

Note: you should never modify objects returned from the shared cache.

The following are the valid values for the

`org.eclipse.persistence.config.HintValues`:

- `TRUE` – retrieve read-only results back from the query;
- `FALSE` – do not retrieve read-only results back from the query.

Default: `FALSE`

Example: JPA Query API

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
query.setHint(QueryHints.READ_ONLY, HintValues.TRUE);
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
@QueryHint(name=QueryHints.READ_ONLY, value=HintValues.TRUE);
```

Result Collection Type

The `eclipselink.result-collection-type` hint configures the concrete class that EclipseLink should use to return its query result.

This lets you specify the type of collection in which the result will be returned.

Valid values: Java `Class` that implements the `Collection` interface.

Note: typically, you would execute these queries by calling the `getResultsList` method, which returns the java.util.List, on the `Query`. This means that the class specified in this hint must implement the `List` interface, if you are invoking it using the `getResultsList` method.

Note: specify the class without the `".class"` notation. For example, `java.util.Vector` would work, not `java.util.Vector.class`EclipseLink will throw an exception, if you use this hint with a class that does not implement the `Collection` interface.

Default:`java.util.Vector`

Example: JPA Query API

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
query.setHint("eclipselink.result-collection-type",
java.util.ArrayList.class);
```

Example: `@QueryHint`

```
import org.eclipse.persistence.config.HintValues;
import org.eclipse.persistence.config.QueryHints;
@QueryHint(name=QueryHints.RESULT_COLLECTION_TYPE,
value="java.util.ArrayList");
```

How to Use EclipseLink Query API in JPA Queries

EclipseLink JPA provides an EclipseLink implementation class for each JPA persistence interface. By casting to the EclipseLink implementation class, you have full access to EclipseLink functionality.

This section provides the following examples:

- Creating a JPA Query Using the EclipseLink Expressions Framework
- Creating a JPA Query Using an EclipseLink DatabaseQuery
- Creating a JPA Query Using an EclipseLink Call Object
- Using Named Parameters in a Native Query
- Using Java Persistence Query Language Positional Parameters in a Native Query
- Using JDBC-Style Positional Parameters in a Native Query

For more information, see Queries.

Creating a JPA Query Using the EclipseLink Expressions Framework

EclipseLink provides an expression framework with which you can express queries in a database-neutral fashion as an alternative to raw SQL.

This example shows how to cast an entity manager to access an EclipseLink persistence provider `createQuery` method that takes an EclipseLink `Expression`.

Creating a Query with the EclipseLink Expressions Framework

```
((org.eclipse.persistence.jpa.JpaEntityManager)entityManager.getDe
legate()).
                createQuery(Expression expression, Class
resultType);
```

EclipseLink 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.
  For example, the Java code required to get the street name from the `Address` object of the `Employee` class looks like this:

```
emp.getAddress().getStreet().equals("Meadowlands");
```

The expression to get the same information is similar:

```
emp.get("address").get("street").equal("Meadowlands");
```

- 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.
  For example, the following Java code retrieves all employees that live on "Meadowlands" whose salary is greater than 10,000:

```
ExpressionBuilder emp = new ExpressionBuilder();
Expression exp =
emp.get("address").get("street").equal("Meadowlands");
Vector employees = session.readAllObjects(Employee.class,
exp.and(emp.get("salary").greaterThan(10000)));
```

EclipseLink automatically generates the appropriate SQL from the preceding code:

```
SELECT t0.VERSION, t0.ADDR_ID, t0.EMP_ID, t0.SALARY FROM EMPLOYEE
t0, ADDRESS t1 WHERE (((t1.STREET = 'Meadowlands')AND (t0.SALARY >
10000)) AND (t1.ADDRESS_ID = t0.ADDR_ID))
```

For more information, see Introduction to EclipseLink Expressions.

Creating a JPA Query Using an EclipseLink DatabaseQuery

An EclipseLink `DatabaseQuery` is a query object that provides a rich API for handling a variety of database query requirements, including reading and writing at the object level and at the data level.

This example shows how to cast a JPA query from an entity manager to access an EclipseLink persistence provider `setDatabaseQuery` method that takes an EclipseLink `DatabaseQuery`.

DatabaseQuery

```
((org.eclipse.persistence.jpa.JpaQuery)query).setDatabaseQuery(Dat
abaseQuery query);
```

The following example shows how to cast a JPA query from an entity manager to access an EclipseLink persistence provider `setDatabaseQuery` method that takes an EclipseLink `DataReadQuery` initialized with an EclipseLink `SQLCall` object that specifies a `SELECT`. This query will return one or more objects.

DatabaseQuery with Selecting Call

```
((org.eclipse.persistence.jpa.JpaQuery)query).
   setDatabaseQuery(new DataReadQuery(new SQLCall("SELECT...")));
```

The following example shows how to cast a JPA query from an entity manager to access an EclipseLink persistence provider `setDatabaseQuery` method that takes an EclipseLink `DataModifyQuery` initialized with an EclipseLink `SQLCall` object that specifies an `UPDATE`. This query will modify one or more objects; however, this query will not update the managed objects within the persistence context.

DatabaseQuery with Non-Selecting Call

```
((org.eclipse.persistence.jpa.JpaQuery)query).
      setDatabaseQuery(new DataModifyQuery(new
SQLCall("UPDATE...")));
```

Creating a JPA Query Using an EclipseLink Call Object

Using `DatabaseQuery` method `setCall`, you can define your own EclipseLink `Call` to accommodate a variety of data source options, such as SQL stored procedures and stored functions, EJB QL queries, and EIS interactions.

This example shows how to cast a JPA query from an entity manager to access an EclipseLink persistence provider `getDatabaseQuery` method to set a new `SQLCall`.

Call

```
((org.eclipse.persistence.jpa.JpaQuery)query).
            getDatabaseQuery().setCall(new SQLCall("..."));
```

For more information, see Call Queries.

Using Named Parameters in a Native Query

Using EclipseLink, you can specify a named parameter in a native query using the EclipseLink `#` convention (see the Specifying a Named Parameter with # example).

Support for the EclipseLink `#` convention is helpful if you are already familiar with EclipseLink queries or if you are migrating EclipseLink queries to a JPA application.

Specifying a Named Parameter with #

```
Query queryEmployees = entityManager.createNativeQuery(
   "SELECT * FROM EMPLOYEE emp WHERE emp.fname LIKE #firstname");
queryEmployees.setParameter("firstName", "Joan");
Collection employees = queryEmployees.getResultList();
```

Using JP QL Positional Parameters in a Native Query

Using EclipseLink, you can specify positional parameters in a native query using the Java Persistence query language (JP QL) positional parameter convention `?n` to specify a parameter by number. For more information on JP QL, see What You May Need to Know About Querying with Java Persistence Query Language.

This example shows how to specify positional parameters using the `?n` convention. In this example, the query string will be `SELECT * FROM EMPLOYEE WHERE F_NAME LIKE "D%" AND L_NAME LIKE "C%"`.

Specifying Positional Parameters Using ?

```
Query queryEmployees = entityManager.createNativeQuery(
   "SELECT * FROM EMPLOYEE WHERE F_NAME LIKE?1 AND L_NAME LIKE?2",
Employee.class);
queryEmployees.setParameter(1, "D%");
queryEmployees.setParameter(2, "C%");
Collection employees = queryEmployees.getResultList();
```

You can easily re-use the same parameter in more than one place in the query, as the following example shows. In this example, the query string will be `SELECT * FROM EMPLOYEE WHERE F_NAME LIKE "D%" AND L_NAME LIKE "D%"`.

Specifying Positional Parameters Using ?n

```
Query queryEmployees = entityManager.createNativeQuery(
   "SELECT * FROM EMPLOYEE WHERE F_NAME LIKE?1 AND L_NAME LIKE?1",
Employee.class);
queryEmployees.setParameter(1, "D%");
Collection employees = queryEmployees.getResultList();
```

Using JDBC-Style Positional Parameters in a Native Query

Using EclipseLink, you can specify positional parameters in a native query using the JDBC-style positional parameter `?` convention.

This example shows how to specify positional parameters using the `?` convention. Each occurrence of `?` must be matched by a corresponding `setParameter` call. In this example, the query string will be `SELECT * FROM EMPLOYEE WHERE F_NAME LIKE "D%" AND L_NAME LIKE "C%"`.

Specifying Positional Parameters with ?

```
Query queryEmployees = entityManager.createNativeQuery(
   "SELECT * FROM EMPLOYEE WHERE F_NAME LIKE? AND L_NAME LIKE?",
Employee.class);
queryEmployees.setParameter(1, "D%");
queryEmployees.setParameter(2, "C%");
Collection employees = queryEmployees.getResultList();
```

If you want to re-use the same parameter in more than one place in the query, you must repeat the same parameter, as this example shows. In this example, the query string will be `SELECT * FROM EMPLOYEE WHERE F_NAME LIKE "D%" AND L_NAME LIKE "D%"`.

Re-Using Positional Parameters with ?

```
Query queryEmployees = entityManager.createNativeQuery(
   "SELECT * FROM EMPLOYEE WHERE F_NAME LIKE? AND L_NAME LIKE?",
Employee.class);
queryEmployees.setParameter(1, "D%");
queryEmployees.setParameter(2, "D%");
Collection employees = queryEmployees.getResultList();
```

Using EclipseLink JPA Weaving

Weaving is a technique of manipulating the byte-code of compiled Java classes. The EclipseLink JPA persistence provider uses weaving to enhance JPA entities for such things as lazy loading, change tracking, fetch groups, and internal optimizations.

This section describes the following:

How to Configure Dynamic Weaving for JPA Entities Using the EclipseLink Agent

Use this option to weave applicable 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 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. For more information, see How to Configure Static Weaving for JPA Entities.

To Configure Dynamic Weaving for JPA Entities Using the EclipseLink Agent

- Configure your `persistence.xml` file with a `eclipselink.weaving` extension set to `true`, as this example shows.
  Setting eclipselink.weaving in the persistence.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
      <persistence-unit name="HumanResources">
           <provider>org.eclipse.persistence.jpa.Persistence
Provider</provider>
           <class>com.acme.Employee</class>
           <!-- ... -->
           <properties>
                <property
                     name="eclipselink.weaving"
                     value="true"
                />
           </properties>
      </persistence-unit>
```

```
</persistence>
```

For more information, see the EclipseLink JPA Persistence Unit Properties for Customization and Validation table.

- Modify your application JVM command line to include the following:

```
-javaagent:eclipselink.jar
```

- Ensure that the `eclipselink.jar` is in your application classpath.
- Package and deploy your application.
  For more information, see Packaging an EclipseLink JPA Application.

EclipseLink weaves applicable class files one at a time, as they are loaded at run time.

How to Configure Static Weaving for JPA Entities

Use this option to weave all applicable class files at build time so that you can deliver pre-woven 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 (see How to Configure Dynamic Weaving for JPA Entities Using the EclipseLink Agent).

In addition, consider using this option to weave in Java environments where you cannot configure an agent.

Prior to weaving, your persistence unit should be set-up in a way that is understood by eclipselink. There are two basic configurations:

1. A jar file - setup as specified in the JPA specification

- classes stored at the base in directories based on their package structure
- a META-INF directory containing your persistence.xml. Note: Using the persistenceunitinfo setting below, you can avoid this requirement

e.g. mypersistenceunit.jar could contain

- *mypackage/MyEntity1.class*
- *mypackage/MyEntity2.class*
- *mypackage2/MyEntity3.class*
- *META-INF/persistence.xml*

2. An exploded directory structure

- classes stored at the base in directories based on their package structure
- a META-INF directory containing your persistence.xml. Note: Using the persistenceunitinfo setting below, you can avoid this requirement

e.g. If your base directory was c:/classes, the exploded directory structure would look as follows:

- *c:/classes/mypackage/MyEntity1.class*
- *c:/classes/mypackage/MyEntity2.class*
- *c:/classes/mypackage2/MyEntity3.class*

- *c:/classes/META-INF/persistence.xml*

To Configure Static Weaving for JPA Entities

- Execute the static static weaver in one of the following ways:
  - Use the `weave` Ant task as follows:
    - Configure the `weave` Ant task in your build script, as this example shows. The EclipseLink weave Ant Task Attributes table lists the attributes of this task.

EclipseLink weave Ant Task

```
<target name="define.task" description="New task
definition for EclipseLink static weaving"/>

   <taskdef name="weave"
classname="org.eclipse.persistence.tools.weaving.j
pa.StaticWeaveAntTask"/>
</target>
<target name="weaving" description="perform
weaving" depends="define.task">
   <weave  source="c:\myjar.jar"
         target="c:\wovenmyjar.jar"
         persistenceinfo="c:\myjar-containing-
persistenceinfo.jar">
      <classpath>
         <pathelement path="c:\myjar-
dependent.jar"/>
      </classpath>

   </weave>
</target>
```

EclipseLink `weave` Ant Task Attributes

Attribute	Description	Default
`source`	Specifies the location of the Java source files to weave: either a directory or a JAR file. If the `persistence.xml` file is not in a META-INF directory at this location, you must specify the location of the `persistence.xml` using the `persistenceinfo` attribute.	
`target`	Specifies the output location: either a directory or a JAR file.	

`persistenceinfo`	Specifies the location of the `persistence.xml` file if it is not in the same location as the source. Note: persistence.xml should be put in a directory called META-INF at this location	
`log`	Specifies a logging file.	See Loggin
`loglevel`	Specifies the amount and detail of log output. Valid `java.util.logging.Level` values are the following: • `OFF` • `SEVERE` • `WARNING` • `INFO` • `CONFIG` • `FINE` • `FINER` • `FINEST` For more information, see Logging.	`Level.OF`

Note: If `source` and `target` point to the same location and, if the `s` is a directory (not a JAR file), EclipseLink will weave in place. If `sou` and `target` point to different locations, or if the `source` is a JAR fil the EclipseLink weave Ant Task example shows), EclipseLink canno weave in place.

- Configure the `weave` task with an appropriate `<classpath>` element, as the EclipseLink weave Ant Task example shows, so that EclipseLink can load all required source classes.
- Execute the Ant task using the command line that this example shows.
  In this example, the `weave` Ant task is in the `build.xml` file:
  EclipseLink weave Ant Task Command Line

```
ant -lib C:\eclipselink.jar -f build.xml weave
```

Note: You must specify the `eclipselink.jar` file (the JAR that c the EclipseLink `weave` Ant task) using the Ant command line `-lib`

instead of using the `taskdef` attribute `classpath`.

- Use the command line as follows:

```
java
org.eclipse.persistence.tools.weaving.jpa.StaticWeave
[arguments] <source> <target>
```

The following example shows how to use the `StaticWeave` class on Windows systems. The EclipseLink StaticWeave Class Command Line Arguments table lists the arguments of this class.

Executing StaticWeave on the Command Line

```
java
org.eclipse.persistence.tools.weaving.jpa.StaticWeave
-persistenceinfo c:\myjar-containing-persistencexml.jar
-classpath c:\classpath1;c:\classpath2 c:\myjar-
source.jar c:\myjar-target.jar
```

EclipseLink StaticWeave Class Command Line Arguments

Argument	Description	Default	Re or Op
`-persistenceinfo`	Specifies the location of the `persistence.xml` file if it is not at the same location as the source (see `-classpath`) Note: EclipseLink will look in a META-INF directory at that location for persistence.xml.		Op
`-classpath`	Specifies the location of the Java source files to weave: either a directory or a JAR file. For Windows systems, use delimiter `" ; "`, and for Unix, use delimiter `" : "`. If the `persistence.xml` file is not in this location, you must specify the location of the `persistence.xml` using the `-persistenceinfo` attribute.		Re

`-log`	Specifies a logging file.	See Logging.	O
`-loglevel`	Specifies the amount and detail of log output. Valid `java.util.logging.Level` values are as follows: • `OFF` • `SEVERE` • `WARNING` • `INFO` • `CONFIG` • `FINE` • `FINER` • `FINEST` For more information, see Logging.	`Level.OFF`	O
`<source>`	Specifies the location of the Java source files to weave: either a directory or a JAR file. If the `persistence.xml` file is not in this location, you must specify the location of the `persistence.xml` using the `-persistenceinfo` attribute.		R
`<target>`	Specifies the output location: either a directory or a JAR file.		R

Note: If `<source>` and `<target>` point to the same location and if the `<source>` is a directory (not a JAR file), EclipseLink will weave in place. If `<source>` and `<target>` point to different locations, or if the `source` is a JAR file (as the Executing StaticWeave on the Command Line example shows), EclipseLink cannot weave in place.

- Configure your `persistence.xml` file with a `eclipselink.weaving` extension set to `static`, as this example shows:
  Setting eclipselink.weaving in the persistence.xml File

```
<persistence>
   <persistence-unit name="HumanResources">
```

```
      <class>com.acme.Employee</class>
      ...
      <properties>

         <property
            name="eclipselink.weaving"
            value="static"
         >
      </properties>
   </persistence-unit>
</persistence>
```

For more information, see the EclipseLink JPA Persistence Unit Properties for Customization and Validation table.

- Package and deploy your application.
  For more information, see Packaging and Deploying EclipseLink JPA Applications.

How to Disable Weaving Using EclipseLink Persistence Unit Properties

To disable weaving, you use 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`:
  - `eclipselink.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.

This example shows how to disable weaving for change tracking only.

Disabling Weaving for Change Tracking in the persistence.xml File

```
<persistence>
   <persistence-unit name="HumanResources">
      <class>com.acme.Employee</class>

      ...
      <properties>
         <property
            name="eclipselink.weaving.changetracking"
            value="false"
         >
      </properties>
```

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

The following example shows how to disable all weaving: in this example, EclipseLink does not weave for lazy loading (indirection), change tracking, or internal optimization.

Disabling All Weaving in the persistence.xml File

```
<persistence>
   <persistence-unit name="HumanResources">
      <class>com.acme.Employee</class>

      ...
      <properties>
         <property
            name="eclipselink.weaving"
            value="false"
         >
      </properties>
   </persistence-unit>
</persistence>
```

For more information, see the EclipseLink JPA Persistence Unit Properties for Customization and Validation table.

- Package and and deploy your application. For more information, see Packaging and Deploying EclipseLink JPA Applications.

What You May Need to Know About Weaving JPA Entities

The EclipseLink JPA persistence provider uses weaving to enable the following features for JPA entities:

- lazy loading (indirection): see How to Configure Lazy Loading;
- change tracking: see How to Configure Change Tracking;
- fetch groups: see How to Configure Fetch Groups;
- internal optimizations: see Optimizing the EclipseLink Application.

EclipseLink weaves all the JPA entities in a given persistence unit. That is the following:

- all classes you list in `persistence.xml` file;
- if element `<exclude-unlisted-classes>` is `false`, or deployed in Java EE, all classes in the JAR file containing the `persistence.xml` file;
- all classes you list in the `orm.xml` file.

For more information, see What You May Need to Know About Weaving and Java EE Application Servers.

What You May Need to Know About EclipseLink JPA Lazy Loading

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 Java EE environment, you only have to set fetch to `javax.persistence.FetchType.LAZY`, and EclipseLink persistence provider will supply all the necessary functionality.

When using a one-to-one or many-to-one mapping in a Java SE environment, to configure EclipseLink JPA to perform lazy loading when the `fetch` attribute is set to `FetchType.LAZY`, configure either dynamic or static weaving.

When using a one-to-one or many-to-one mapping in a Java SE environment that does not permit the use of `-javaagent` on the JVM command line, to configure EclipseLink JPA to perform lazy loading when annotation attribute `fetch` is set to `javax.persistence.FetchType.LAZY`, you can use static weaving.

The EclipseLink JPA Support for Lazy Loading by Mapping Type table lists EclipseLink JPA support for lazy loading by mapping type.

For more information, see the following:

- How to Configure Dynamic Weaving for JPA Entities Using the EclipseLink Agent
- How to Configure Static Weaving for JPA Entities
- How to Disable Weaving Using EclipseLink Persistence Unit Properties
- What You May Need to Know About Weaving JPA Entities
- Configuring Indirection (Lazy Loading)

EclipseLink JPA Support for Lazy Loading by Mapping Type

Mapping	Java EE [1]	Java SE
many-to-many	EclipseLink JPA performs lazy loading when the `fetch` attribute is set to `javax.persistence.FetchType.LAZY` (default).	EclipseLink JPA performs lazy the `fetch` attribute is set to `javax.persistence.Fetc` (default).
one-to-many	EclipseLink JPA performs lazy loading when the `fetch` attribute is set to `javax.persistence.FetchType.LAZY` (default).	EclipseLink JPA performs lazy the `fetch` attribute is set to `javax.persistence.Fetc` (default).
one-to-one	EclipseLink JPA performs lazy loading when the `fetch` attribute is set to `javax.persistence.FetchType.LAZY`.	By default, EclipseLink JPA ig attribute and default javax.persistence.`FetchType` To configure EclipseLink JPA loading when the `fetch` attrib `FetchType.LAZY`, consider

		following: • How to Configure Dyna JPA Entities Using the • How to Configure Stati JPA Entities
many-to-one	EclipseLink JPA performs lazy loading when the `fetch` attribute is set to `javax.persistence.FetchType.LAZY`.	By default, EclipseLink JPA ig attribute and default `javax.persistence.Fetc` applies. To configure EclipseLink JPA loading when the `fetch` attrib `FetchType.LAZY`, configure following: • How to Configure Dyna JPA Entities Using the • How to Configure Stati JPA Entities
basic	EclipseLink JPA performs lazy loading when the `fetch` attribute is set to `javax.persistence.FetchType.LAZY`.	By default, EclipseLink JPA ig attribute and default javax.persistence.`FetchType` To configure EclipseLink JPA loading when the `fetch` attrib `FetchType.LAZY`, consider following: • How to Configure Dyna JPA Entities Using the • How to Configure Stati JPA Entities

[1] Fully supported in any container that implements the appropriate container contracts in the EJB 3.0 specification.

What You May Need to Know About Overriding Annotations in JPA

In JPA, you override any annotation with XML in your object relational mapping files (see Overriding Annotations with XML).

In EclipseLink JPA, you either use JPA processing, or you specify the `sessions.xml` file resulting in creation of the `project.xml` file. For more information, see What You May Need to Know About EclipseLink JPA Overriding Mechanisms.

Overriding Annotations with eclipselink-orm.xml File

The eclipselink-orm.xml file is the EclipseLink native metadata XML file. It can be used to override the JPA configurations defined in the JPA orm.xml file. This exclipselink-orm.xml file provides access to the the advanced features provided by the EclipseLink XSD (http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_1_0.xsd).

See this page for more information and examples.

Overriding Annotations with XML

In JPA, you can use XML mapping metadata on its own, or in combination with annotation metadata, or you can use it to override the annotation metadata.

If you choose to include one or more mapping XML files in your persistence unit, each file must conform and be valid against the `orm_1_0.xsd` schema located at `http://java.sun.com/xml/ns/persistence/orm_1_0.xsd`. This schema defines a namespace called `http://java.sun.com/xml/ns/persistence/orm` that includes all of the ORM elements that you can use in your mapping file.

All object relational XML metadata is contained within the `entity-mappings` root element of the mapping file. The subelements of `entity-mappings` can be categorized into four main scoping and functional groups: persistence unit defaults, mapping file defaults, queries and generators, and managed classes and mappings. There is also a special setting that determines whether annotations should be considered in the metadata for the persistence unit (see Disabling Annotations).

For more information and examples, see Section 10.1 "XML Overriding Rules" of the JPA Specification.

Disabling Annotations

JPA provides a mechanism that you can use to disable annotations. If you do not feel the need for annotations in your application, you can use the `xml-mapping-metadata-complete` and `metadata-complete` mapping file elements to disable any existing annotations. Setting this options causes the processor to completely ignore annotations.

When you specify the `xml-mapping-metadata-complete` element, all annotations in the persistence unit will be ignored, and only mapping files in the persistence unit will be considered as the total set of provided metadata. Only entities (see Section 8.1 "Entity" of the JPA Specification), mapped superclasses (see @MappedSuperclass), and embedded objects (see @Embedded) that have entries in a mapping file will be added to the persistence unit. The `xml-mapping-metadata-complete` element has to be in only one of the mapping files in the persistence unit. You specify it as an empty subelement of the `persistence-unit-metadata` element, as this example shows:

Disabling Annotations for the Persistence Unit in the Mapping File

```
<entity-mappings>
   <persistence-unit-metadata>
      <xml-mapping-metadata-complete/>
   </persistence-unit-metadata>
   ...
```

```
</entity-mappings>
```

You can also use the `metadata-complete` attribute of the `entity`, `mapped-superclass`, and `embeddable` elements. If you specify this attribute, all annotations on the specified class and on any fields or properties in the class will be ignored – only metadata in the mapping file will be considered as the set of metadata for the class. However, even though the annotations are ignored, the default mapping still applies, so any fields or properties that should not be mapped must still be marked as transient in the XML file, as this example demonstrates.

Disabling Annotations for a Managed Class in the Mapping File

```
@Entity
 public class Employee {

       @Id
       private int id;

       @Column(name="EMP_NAME")
       private String name;

       @Column(name="SALARY")
       private long salary;

       ...
 }

<entity-mappings>
       ...
       <entity class="mypackage.Employee" '''metadata-
complete="true"''''>
            <attributes>
                 <id name="id"/>

            </attributes>
       </entity>
       ...
 </entity-mappings>
```

In the preceding example, the entity mappings in the annotated class are disabled by the `metadata-complete` attribute, and because the fields are not mapped in the mapping file, the default mapping values will be used. The `name` and `salary` fields will be mapped to the `NAME` and `SALARY` columns, respectively.

For more information, see Section 10.1 "XML Overriding Rules" of the JPA Specification.

Advantages and Disadvantages of Using Annotations

Metadata annotations are relatively simple to use and understand. They provide in-line

metadata located with the code that this metadata is describing – you do not need to replicate the source code context of where the metadata applies.

On the other hand, annotations unnecessarily couple the metadata to the code. Thus, changes to metadata require changing the source code.

Advantages and Disadvantages of Using XML

The following are the advantages of using XML:

- 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 are the complexity and the need for replication of the code context.

What You May Need to Know About EclipseLink JPA Overriding Mechanisms

EclipseLink JPA provides a set of persistence unit properties (see What you May Need to Know About Using EclipseLink JPA Persistence Unit Properties) that you can specify in your `persistence.xml` file. The persistence unit properties always override the corresponding annotations' attributes.

Similar to EclipseLink annotations, properties expose some features of EclipseLink that are currently not available through the use of JPA metadata.

Note: If multiple instances of the same property are set, then EclipseLink will use the values from the last entry in the list. However, the properties Map provided in the `createEntityManagerFactory` method will always have precedence.

You can also specify the persistence information in the EclipseLink session configuration file – `sessions.xml` (see Session Configuration and the sessions.xml File). By setting the `eclipselink.sessions-xml` persistence unit property you enable EclipseLink to replace all class annotations and object relational mappings that you defined in the `persistence.xml` file, as well as mapping files (if present). Through the `sessions.xml` file the `eclipselink.sessions-xml` property lets you provide session-level configurations that are not supported by persistence unit properties (for example, cache coordination).

Note: You can use the `eclipselink.sessions-xml` property as an alternative to annotations and deployment XML.

For more information on creating and configuring the `sessions.xml` file, see the following:

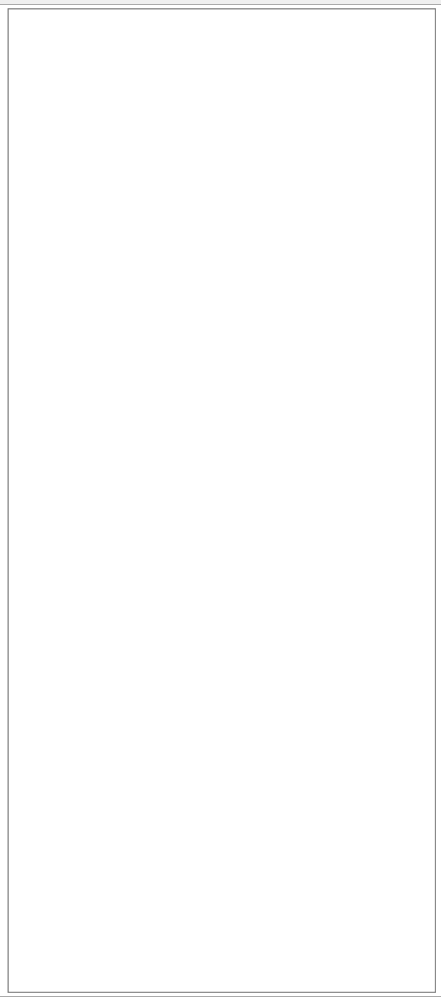
- Acquiring a Session at Run Time with the Session Manager
- Building and Using the Persistence Layer
- Loading project.xml or sessions.xml Files
- Development Environment
- Introduction to the EclipseLink Deployment File Creation
- Packaging an EclipseLink Application
- EclipseLink Sessions XML File
- Introduction to the Session Creation
- Creating a Sessions Configuration
- Configuring a Sessions Configuration
- Introduction to Session Acquisition

In summary, EclipseLink JPA possesses an overriding mechanism that you can use in the following ways:

- You can combine the use of JPA annotations, object relational mapping files (such as `orm.xml`) and persistence unit properties.In this case, EclipseLink persistence provider builds metadata starting with applying defaults, then JPA annotations, and then overrides that with elements of the object relational mapping file. This results in creation of an in-memory EclipseLink session and project. Then EclipseLink persistence provider applies persistence unit properties specified in `persistence.xml` file, as the following illustration shows.

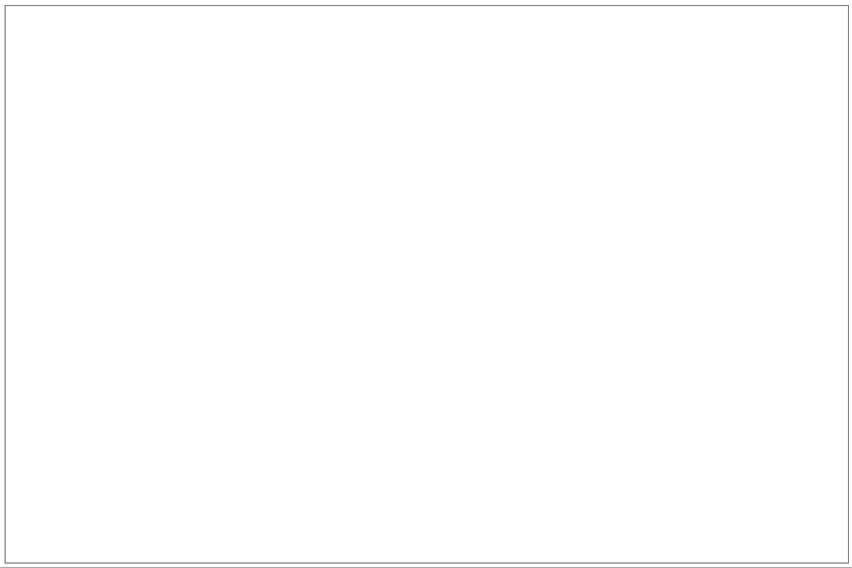
Note: The eclipselink.sessions-xml property represents a special case discussed further.

Combining the Use of Annotations, orm.xml File and Persistence Unit Properties



- You can use the `eclipselink.sessions-xml` persistence unit property. This defines a `sessions.xml` file, which references a `project.xml` file. In this case, EclipseLink persistence provider builds an in-memory EclipseLink session and project based on this metadata, as the following illustration shows. You can acquire a persistence manager and use it as per the JPA specification, having defined all entities and so on using only EclipseLink `sessions.xml` (see Creating Session Metadata) and `project.xml` files that you created Workbench (see Creating the project.xml File with Workbench|Creating the project.xml File with Workbench).

Using eclipselink.sessions-xml Persistence Unit Property



Note: You cannot combine the use of JPA annotations, object relational mapping files, persistence unit properties and the `eclipselink.sessions-xml` persistence unit property.

What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties

A persistence unit configures various details that are required when you acquire an entity manager. You specify a persistence unit by name when you acquire an entity manager factory.

You configure persistence units in JPA persistence descriptor file `persistence.xml`.

In this file, you can specify the vendor extensions that this reference describes by using a `<properties>` element. The Configuring a Vendor Extension in the persistence.xml File (Java EE) example shows how to set an EclipseLink JPA persistence unit extension in a `persistence.xml` file for a Java EE application. The Configuring a Vendor Extension in the persistence.xml File (Java SE) example shows how to do the same for a Java SE application.

Note: that EclipseLink does not provide a mechanism for encrypting the password in the `persistence.xml` file. (see Avoiding Cleartext Passwords)

Configuring a Vendor Extension in the persistence.xml File (Java EE)

```
<persistence-unit name="default" transaction-type="JTA">
      <provider>
           org.eclipse.persistence.jpa.PersistenceProvider
      </provider>
      <jta-data-source>
           jdbc/MyDataSource
      </jta-data-source>

      <properties>
           <property name="eclipselink.logging.level"
value="INFO"/>
      </properties>
 </persistence-unit>
```

Configuring a Vendor Extension in the persistence.xml File (Java SE)

```
<persistence-unit name="default" transaction-
type="RESOURCE_LOCAL">
      <provider>
           org.eclipse.persistence.jpa.PersistenceProvider
      </provider>
```

```
      <exclude-unlisted-classes>false</exclude-unlisted-classes>
      <properties>
           <property name="eclipselink.logging.level"
value="INFO"/>
           <property name="eclipselink.jdbc.driver"
value="oracle.jdbc.OracleDriver"/>
           <property name="eclipselink.jdbc.url"
value="jdbc:oracle:thin:@myhost:1521:MYSID"/>
           <property name="eclipselink.jdbc.password"
value="tiger"/>
           <property name="eclipselink.jdbc.user" value="scott"/>
      </properties>

 </persistence-unit>
```

Alternatively, you can set a vendor extensions in the `Map` of properties you pass into a call to `javax.persistence.Persistence` method `createEntityManagerFactory`, as the Configuring a Vendor Extension when Creating an EntityManagerFactory example shows. You can override extensions set in the `persistence.xml` file in this way. When you set an extension in a map of properties, you can set the value using the public static final field in the appropriate configuration class in `org.eclipse.persistence.config`, including the following:

- `CacheType`
- `TargetDatabase`
- `TargetServer`
- `PersistenceUnitProperties`

The following example shows how to set the value of extension `eclipselink.cache.type.default` using the `CacheType` configuration class.

Configuring a Vendor Extension when Creating an EntityManagerFactory

```
import org.eclipse.persistence.config.CacheType;

Map properties = new HashMap();
properties.put(PersistenceUnitProperties.CACHE_TYPE_DEFAULT,
CacheType.Full);
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("default", properties);
```

You may specify any EclipseLink JPA extension in a `persistence.xml` file and you may pass any EclipseLink JPA extension into `Persistence` method `createEntityManagerFactory`.

Currently, no EclipseLink JPA extensions are applicable to `EntityManagerFactory` method `createEntityManager`.

For more information, see the following:

- Section 2.1 "Requirements on the Entity Class" of the JPA Specification;
- Chapter 5 "Entity Managers and Persistence Contexts" of the JPA Specification.

Allowing Zero Value Primary Keys

By default, EclipseLink interprets zero as *null* for primitive types that cannot be null (such as `int` and `long`) causing zero to be an invalid value for primary keys. You can modify this setting by using the **allow-zero-id property** in the `persistence.xml file`. Valid values are:

- **true** – EclipseLink interprets zero values as *zero*. This permits primary keys to use a value of zero.
- **false** (default) – EclipseLink interprets zero as *null*.

Avoiding Cleartext Passwords

EclipseLink does not support storing encrypted passwords in the `persistence.xml` file. For a Java EE application, you do not need to specify your password in the `persistence.xml` file. Instead, you can specify a `data-source`, as shown in the Java EE example. This datasource is specified on the application server, and can encrypt the your password with its own mechanism.

For a Java SE application, you should avoid putting the password in the `persistence.xml` file. Instead, add the password to the `properties`, as shown in the Setting the password in code example. Setting the password in code allows for the password to be retrieved from any location, including a protected source.

Setting the password in code

```
Map properties = new HashMap();
 properties.put(PersistenceUnitProperties.JDBC_PASSWORD, "tiger");
 EntityManagerFactory emf =
Persistence.createEntityManagerFactory("default", properties);
```


Configuring a EclipseLink JPA Application

Related Topics

This section contains information on how you can configure your EclipseLink persistence project.

When configuring an EclipseLink JPA application you cannot use the `org.eclipselink.sessions.Project` class, but you can still customize sessions and descriptors by using the customizer extensions (see Using EclipseLink JPA Extensions for Customization and Optimization).

You can set up your EclipseLink JPA application using various IDEs.

At run time, if the `eclipselink.jar` file is on the application classpath, you have the option to choose EclipseLink as a persistence provider for your application.

Configuring Oracle Database Proxy Authentication for a JPA Application

One of the features of the Oracle Database is proxy authentication. For more information, see Oracle Database Proxy Authentication.

How to Provide Authenticated Reads and Writes of Secured Data Through the Use of an Exclusive Isolated Client Session

If you use Oracle Virtual Private Database (VPD) (see Isolated Client Sessions and VPD), and you want to enable read and write access control for your EclipseLink JPA application, in the SessionCustomizer class set the connection policy to use exclusive connections, and define the descriptor for secured data as isolated (see Configuring Cache Isolation at the Descriptor Level). Consider the following example.

Pass properties to the `createEntityManagerFactory` method, as the following example demonstrates. One property should indicate that exclusive connection should be used for classes that use isolated cache. Other properties should indicate that one or more specific entities use an isolated cache,

```
Map properties = new HashMap();
properties.put("eclipselink.jdbc.exclusive-connection.mode",
"Isolated");
properties.put("eclipselink.cache.shared.Employee", "false");
...
EntityManagerFactory emf =
Persistence.createEntityManagerFactory(properties);
```

In the preceding example, an entity named `Employee` uses isolated cache and will be read and written through an exclusive connection.

To specify that all entities are to use isolated cache, set the `eclipselink.cache.shared.default` property to `false`:

```
properties.put("eclipselink.cache.shared.default", "false");
```

For more information, see How to Use the Persistence Unit Properties for Caching.

How to Provide Authenticated Writes for Database Auditing Purposes with a Client Session

If you want to enable write access control for your EclipseLink JPA application, in your SessionCustomizer class provide the `EntityManager` with properties similar to the ones that the following example demonstrates.

```
Map emProperties = new HashMap();
emProperties.put("eclipselink.oracle.proxy-type",
oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
emProperties.put(oracle.jdbc.OracleConnection.PROXY_USER_NAME,
"john");
EntityManager em = emFactory.createEntityManager(emProperties);
```

In the preceding example, the `EntityManager` uses a proxy user "john" for writes and reads inside a transaction. Note that reads, which are performed outside of the transaction, are done through the main (non-proxied) connection.

If you created your `EntityManager` using injection, set the properties as follows:

```
((org.eclipse.persistence.internal.jpa.EntityManagerImpl)em.getDel
egate()).setProperties(emProperties);
```

JPA 2.0 defines a new setProperty method on EntityManager that could be used instead:

```
em.setProperty("eclipselink.oracle.proxy-type",
oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
em.SetProperty(oracle.jdbc.OracleConnection.PROXY_USER_NAME,
"john");
```

If proxy authentication properties are set when active persistence context already exists (that may happen if setProperty/setProperties method is used) then they will be ignored until a new one is created. Calling clear method on the EntityManager forces creation of the new active persistence context.

For more information, see How to Use the Persistence Unit Properties for Caching.

How to Define Proxy Properties Using EntityManagerFactory

You can also define proxy properties using the `EntityManagerFactory`. If you choose to do so, note that all connections will use these properties, unless they are overridden in the `EntityManager`. Consider the following example:

```
Map factoryProperties = new HashMap();
factoryProperties.put("eclipselink.oracle.proxy-type",
oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
factoryProperties.put(oracle.jdbc.OracleConnection.PROXY_USER_NAME
, "sarah");
EntityManagerFactory emf =
Persistence.createEntityManagerFactory(factoryProperties);

// em1 does not specify its own proxy properties -
// it uses proxy user "sarah" specified by the factory
EntityManager em1 = emf.createEntityManager();

Map emProperties = new HashMap();
emProperties.put("eclipselink.oracle.proxy-type",
oracle.jdbc.OracleConnection.PROXYTYPE_USER_NAME);
emProperties.put(oracle.jdbc.OracleConnection.PROXY_USER_NAME,
"john");

// em2 uses its own proxy properties (proxy user "john"),
// regardless of whether or not factory has proxy properties
EntityManager em2 = emf.createEntityManager(emProperties);

// em3 does not use any proxy connection.
// It cancels proxy properties defined in the factory
Map cancelProperties = new HashMap();
cancelProperties.put("eclipselink.oracle.proxy-type", "");
EntityManager em3 = emf.createEntityManager(cancelProperties);
```

Proxy Authentication and application servers

Oracle Proxy Authentication may have conflicts with application servers' connections that wrap the actual Oracle connection. On some application servers after the proxy session was closed the statements that used the proxy session were still kept in the statement cache - that resulted in exception on attempt to use a closed statement. To workaround set application server statement cache size to zero.

Setting Up Packaging

You can package your application manually (see Packaging an EclipseLink JPA Application), or use an IDE.

Related Topics

Using Application Components

When developing an application with EclipseLink JPA, you need to know how to use the following application components:

- Entity manager factory
- Entity manager
- Persistence context

How to Obtain an Entity Manager Factory

How you obtain the entity manager factory depends on the Java environment in which you are developing your application:

- Obtaining an Entity Manager Factory in Java EE Application Server Environment
- Obtaining an Entity Manager Factory in Java SE Environment

Obtaining an Entity Manager Factory in Java EE Application Server Environment

You can inject an entity manager factory using the `@PersistenceUnit` annotation, as the following example shows, or you can obtain it through JNDI lookup. You may choose to specify the `unitName` element to designate the persistence unit whose factory you are using.

```
@PersistenceUnit
EntityManagerFactory emf;
```

For more information, see the following:

- Section 8.4.2 "PersistenceUnit Annotation" of the JPA Specification
- Section 5.3.1 "Obtaining an Entity Manager Factory in a Java EE Container" of the JPA Specification
- Container-Managed Entity Manager

Obtaining an Entity Manager Factory in Java SE Environment

In Java SE environment, use the `javax.persistence.Persistence` bootstrap class to get access to an entity manager factory. In your application, create an entity manager factory by calling the `javax.persistence.Persistence` class' `createEntityManagerFactory` method (see Section 7.2.1 "javax.persistence.Persistence Class" of the JPA Specification), as the following example shows:

```
EntityManagerFactory emf =
javax.persistence.Persistence.createEntityManagerFactory("Order");
 EntityManager em = emf.createEntityManager();
```

For more information, see the following:

- Section 7.2.1 "javax.persistence.Persistence Class" of the JPA Specification
- Section 5.3.2 "Obtaining an Entity Manager Factory in a Java SE Container" of the JPA Specification
- Application-Managed Entity Manager

How to Obtain an Entity Manager

All entity managers come from factories of type `EntityManagerFactory`. The configuration for an entity manager is bound to the `EntityManagerFactory` that created it, but it is defined separately as a persistence unit. A persistence unit dictates either implicitly or explicitly the settings and entity classes used by all entity managers obtained from the unique `EntityManagerFactory` instance bound to that persistence unit. There is, therefore, a one-to-one correspondence between a persistence unit and its concrete `EntityManagerFactory`.Persistence units are named to allow differentiation of one `EntityManagerFactory` from another. This gives the application control over which configuration or persistence unit is to be used for operating on a particular entity.

How you obtain the entity manager and its factory depends on the Java environment in which you are developing your application:

- Obtaining an Entity Manager in Java EE Application Server Environment
- Obtaining an Entity Manager in Java SE Environment

Obtaining an Entity Manager in Java EE Application Server Environment

In the Java EE environment, you can inject an entity manager using the `@PersistenceContext` annotation, as the following example shows, or you can obtain it through a direct JNDI lookup. You may choose to specify the `unitName` element of the `@PersistenceContext` annotation to designate the persistence unit whose factory the container is using (see Section 8.4.2 "PersistenceUnit Annotation" of the JPA Specification). You can also specify the `type` element to indicate whether a transaction-scoped (default) or extended persistence context is to be used (see Section 5.6 "Container-managed Persistence Contexts" of the JPA Specification).

```
@PersistenceContext
 EntityManager em;

 @PersistenceContext(type=PersistenceContextType.EXTENDED)
 EntityManager orderEM;
```

The container manages the life cycle of the persistence context, as well as the creation and closing of the entity manager–your application does not have to be involved in this process.

For more information, see the following:

- Section 8.4.2 "PersistenceUnit Annotation" of the JPA Specification
- Section 5.6 "Container-managed Persistence Contexts" of the JPA Specification
- Section 5.2.1 "Obtaining an Entity Manager in a Java EE Container" of the JPA Specification
- Container-Managed Entity Manager

Obtaining an Entity Manager in Java SE Environment

You obtain an application-managed entity manager from an entity manager factory.

For more information and examples, see the following:

- Section 5.2.2 "Obtaining an Entity Manager in a Java SE Container" of the JPA Specification
- How to Obtain an Entity Manager Factory
- Application-Managed Entity Manager

What You May Need to Know About Entity Managers and Their Factories

An entity manager persists and manages specific types of objects, enables reading from and writing to a given database. You have to configure the entity manager to do so. You are also responsible for configuring the entity manager to be implemented by a particular persistence provider, such as EclipseLink. The provider supplies the backing implementation engine for the entire Java Persistence API, which includes an entity manager, a `Query` implementation, and SQL generation.

An entity manager implements the API enabling operations on entities. It is encapsulated almost entirely within a single interface called `EntityManager`. Until you use an entity manager to create, read, or write an entity, the entity is nothing more than a regular nonpersistent Java object.

For more information, see Chapter 5 "Entity Managers and Persistence Contexts" of the JPA Specification.

Applications use the `EntityManagerFactory` interface for creating an application-managed entity manager (see Obtaining an Entity Manager in Java SE Environment).

Each entity manager factory provides entity manager instances that are all configured in the same manner (for example, configured to connect to the same database or use the same initial settings as defined by the implementation).

How to Use a Persistence Context

Information pending

Using an Extended Persistence Context

Information pending

What You May Need to Know About Persistence Contexts and Persistence Units

When an entity manager (see What You May Need to Know About Entity Managers and Their Factories) obtains a reference to an entity (either by having it explicitly passed in or because it was read from the database) that object becomes managed by the entity manager. The set of managed entity instances within an entity manager at any given time is called this entity manager's persistence context. Only one Java instance with the same persistent identity may exist in a persistence context at any time. For example, if an Employee with a persistent identity (or id) of 158 exists in the persistence context, then no other object with its id set to 158 may exist within that same persistence context.

An `EntityManager` instance is associated with a persistence context. A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. The entity instances and their life cycle are managed within the persistence context. The `EntityManager` interface defines the methods for interacting with the persistence context. The `EntityManager` API is used to create and remove

persistent entity instances, to find entities by their primary key, and to query over entities.

For more information, see Section 5.1 "Persistence Contexts" of the JPA Specification.

Persistence Unit

The set of entities that a given `EntityManager` instance manages is defined by a persistence unit. A persistence unit defines the set of all classes that are related or grouped by your application, and which must be collocated in their mapping to a single database.

A persistence unit includes the following:

- An entity manager factory and its entity managers, together with their configuration information.
- The set of classes managed by the entity managers.
- Mapping metadata (in the form of metadata annotations and/or XML metadata) that specifies the mapping of the classes to the database.

Querying for an Entity

How to Use the Entity Manager find Method

Information pending

What You May Need to Know About Querying with Java Persistence Query Language

You can use the Java Persistence query language (JP QL) to define queries over entities and their persistent state.

JP QL is an extension of EJB QL, and adds the following features:

- Single and multiple value result types;
- Aggregate functions with sorting and grouping clauses;
- A more natural jon syntax, including support for both inner and outer joins;
- Conditional expressions involving subqueries;
- Update and delete queries for bulk data changes;
- Result projection into nonpersistent classes.

JP QL supports the use of dynamic queries and the use of named parameters. You can use it to define queries over the persistent entities, as well as their persistent state and relationships

You may define queries in metadata annotations or the XML descriptor.

A JP QL statement may be either a select statement, an update statement, or a delete statement. All statement types may have parameters. Any statement may be constructed dynamically or may be statically defined in a metadata annotation or XML descriptor element.

This example demonstrates how to create a simple query that finds all orders using JP QL.

Simple Query to Find All Objects

```
SELECT ORDER
  FROM ORDER ORDER
```

This example demonstrates how to create a simple query that finds all orders to ship to California using JP QL.

Simple Query to Find Some Objects

```
SELECT ORDER
  FROM ORDER ORDER
  WHERE ORDER.shippingAddress.state = 'CA'
```

For more information and examples, see the following:

- Chapter 4 "Query Language" of the JPA Specification
- Section 3.6 "Query API" of the JPA Specification
- What You May Need to Know About Named and Dynamic Queries
- What You May Need to Know About Persisting with JP QL

What You May Need to Know About Named and Dynamic Queries

You can use the `Query` API to define both named and dynamic queries.

Named queries are static and expressed in metadata. You can define named queries using JP QL or SQL, scoping their names to the persistence unit.

Note: The query name must be unique within the scope of the persistence unit.

These queries are efficient to execute as the persistence provider can translate JP QL to SQL once, when you application starts, as opposed to every time the query is executed. You define a named query using the `@NamedQuery` annotation (see Section 8.3.1 "NamedQuery Annotation" of the JPA Specification), which you may place on the class definition for any entity. The annotation defines the name of the query, as well as the query text, as this example shows:

Defining a Named Query

```
@NamedQuery(name="findSalaryForNameAndDepartment",
                query="SELECT e.salary " +
                       "FROM Employee.e " +
                       "WHERE e.department.name =:deptName AND "
+
                       "      e.name =:empName")
```

Place your named query on the entity class that most directly corresponds to the query result. In the preceding example, that would be the `Employee` entity.

If you need to define more than one named query for a class, place them inside of a `@NamedQueries` annotation (see Section 8.3.1 "NamedQuery Annotation" of the JPA Specification) that accepts an array of `@NamedQuery` annotations, as this example shows:

Defining Multiple Named Queries for an Entity

```
@NamedQueries({
      @NamedQuery(name="Employee.findAll",
                      query="SELECT e FROM Employee.e"),
      @NamedQuery(name="Employee.findByPrimaryKey",
                      query="SELECT e FROM Employee.e WHERE e.id
=:id"),
      @NamedQuery(name="Employee.findByName",
                      query="SELECT e FROM Employee.e WHERE e.name
=:name"),
  })
```

Because the query string is defined in the annotation, your application cannot alter it at run time. If you need to specify additional criteria, you must do it using query parameters. This example shows how you can use the `createNamedQuery` method of the `EntityManager` to create a named query that requires a query parameter.

```
''''' Creating a Named Query with Parameters'''''
 @PersistenceContext
 public EntityManager em;
 ...
 customers = em.createNamedQuery("findAllCustomersWithName")
                    .setParameter("custName",
"Smith").getResultList();
```

You may choose to define named queries in an XML mapping file (see Using XML) using the `named-query` element. A `named-query` element in the mapping file may also override an existing query of the same name that was defined as an annotation. A `named-query` element may appear as a subelement of `entity-mapping` or `entity` elements. Regardless of where you defined it, it will be keyed by its name in the persistence unit query namespace. You may provide query hints as `hint` subelements.

This example shows the definition a named query in an XML mapping file. This query uses `eclipselink.cache-usage` hint to bypass the cache.

Defining a Named Query in an XML Mapping File

```
<entity-mapping>
       ...
       <named-query name="findEmployeesWithName">

            <query>SELECT e FROM Employee e WHERE e.name
LIKE:empName</query>
            <hint name="eclipselink.cache-usage"
value="DoNotCheckCache"/>
       </named-query>
       ...
 <entity-mapping>
```

Note: We recommend using named queries with query parameters.

Dynamic queries are strings. You generate these queries at run time by passing the JP QL query string to the `createQuery` method of the `EntityManager`. There are no restrictions on the query definition; all JP QL query types are supported, as well as the use of parameters.

You may consider using dynamic queries in your application, if there might be a need to specify complex criteria and the exact shape of the query cannot be known in advance. However, note that if your application issues many queries, the use of dynamic queries will have a negative impact on performance.

For more information and examples, see the following:

- Section 3.6.4 "Named Queries" of the JPA Specification
- Section 3.6 "Query API" of the JPA Specification
- Using EclipseLink JPA Query Customization Extensions
- Cache
- Named Queries
- What You May Need to Know About Query Hints

Persisting Domain Model Changes

How to Use JTA

Information pending

How to Use RESOURCE_LOCAL

Information pending

How to Configure Flushing and Set Flush Modes

Information pending

How to Manage a Life Cycle of an Entity

Information pending

Merging Detached Entity State

Information pending

Using Detached Entities and Lazy Loading

Information pending

For more information, see the following:

- Section 3.2.4.2 "Detached Entities and Lazy Loading" of JPA specification
- Indirection, Serialization, and Detachment

What You May Need to Know About Persisting with JP QL

You may define queries in metadata annotations or the XML descriptor.

You can use update and delete queries to persist your changes with JP QL.

You can perform bulk update of entities with the `UPDATE` statement. This statement operates on a single entity type and sets one or more single-valued properties of the entity subject to the condition in the `WHERE` clause. Update queries provide an equivalent to the `SQL UPDATE` statement, but with JP QL conditional expressions.

This example demonstrates how to use an update query to give employees a raise. The `WHERE` clause contains the conditional expression.

Update Query

```
UPDATE Employee e
 SET e.salary = 60000
 WHERE e.salary = 50000
```

You can perform bulk removal of entities with the `DELETE` statement. Delete queries

provide an equivalent to the `SQL DELETE` statement, but with JP QL conditional expressions.

This example demonstrates how to use a delete query to remove all employees who are not assigned to a department. The `WHERE` clause contains the conditional expression.

Delete Query

```
DELETE FROM Employee e
 WHERE e.department IS NULL
```

Note: Delete queries are polymorphic: any entity subclass instances that meet the criteria of the delete query will be deleted. However, delete queries do not honor cascade rules: no entities other than the type referenced in the query and its subclasses will be removed, even if the entity has relationships to other entities with cascade removes enabled.

The persistence context is not updated to reflect results of update and delete operations. If you use a transaction-scoped persistence context, you should either execute the bulk operation in a transaction all by itself, or be the first operation in the transaction (see Introduction to EclipseLink Transactions). That is because any entity actively managed by the persistence context will remain unaware of the actual changes occurring at the database level.

For more information and examples, see the following:

- Section 4.10 "Bulk Update and Delete Operations" of the JPA Specification
- Chapter 4 "Query Language" of the JPA Specification
- What You May Need to Know About Querying with Java Persistence Query Language
- Queries
- Named Queries

What You May Need to Know About Persisting Results of Named and Dynamic Queries

Expressions listed in the `SELECT` clause of a query determine the result type of the query. The following are some of the type that may result from JP QL queries:

- Basic types: `String`, primitive types, JDBC types
- Entity types
- An array of `Object` instances
- User-defined types created from a constructor-expressions

The collection or single result corresponds directly to the result type of the query.

The `Query` interface provides three different ways to execute a query, depending on whether or not the query returns results and how many results are expected. For queries that return values, you can call either the following methods:

- `getResultList`–use this method if you expect the query to return more than one result. This method returns a collection (`List`) containing query results. If there are no results to return, this method returns an empty collection.
- `getSingleResult`–use this method if you expect the query to return a single result. In case of unexpected results, such as there are no results to return or multiple results are available, this method throws an exception.

Use the `executeUpdate` method of the `Query` interface to invoke bulk update and delete queries (see What You May Need to Know About Persisting with JP QL).

The active persistence context manages a returned entity instance. If that entity instance is modified and the persistence context is part of a transaction, then the changes will be persisted to the database.

Note: If you use a transaction-scoped entity manager outside of a transaction, then the executed query will return detached entity instances instead of managed entity instances. To make changes to these detached entities, you must merge them into a persistence context before synchronizing with the database.

You can reuse `Query` objects as often as you need so long as the same persistence context that you used to create the query is active. For transaction-scoped entity managers, this limits the lifetime of the `Query` object to the life of the transaction. Other entity manager types may reuse `Query` objects until you close or remove the entity manager.

For more information, see the following:

- Section 3.6 "Query API" of the JPA Specification
- Section 3.6.4 "Named Queries" of the JPA Specification
- Using EclipseLink JPA Query Customization Extensions
- Queries
- Named Queries
- Section 5.6.4.1 "Container-managed Transaction-scoped Persistence Context" of the JPA Specification

Using EclipseLink JPA Extensions in Your Application Development

This section describes the following:

- How to Use Extensions for Query
- How to Configure Lazy Loading
- How to Configure Change Tracking
- How to Configure Fetch Groups
- What You May Need to Know About EclipseLink Caching
- What You May Need to Know About EclipseLink Caching
- What You May Need to Know About Cache Coordination
- How to Configure Cascading

- What You May Need to Know About Cascading Entity Manager Operations
- How to Use EclipseLink Metadata
- How to Use Events and Listeners
- What You May Need to Know About Database Platforms
- What You May Need to Know About Server Platforms
- How to Optimize a JPA Application
- How to Perform Diagnostics

How to Use Extensions for Query

Information pending

Using Query Hints

Information pending

What You May Need to Know About Query Hints

Query hints are the JPA extension point for vendor-specific query features. Hints are the only feature in the query API that are not a standard usage: a hint is a string name and object value.

You may associate your queries with hints by either setting them in the persistence unit metadata as part of the `@NamedQuery` annotation (see Section 8.3.1 "NamedQuery Annotation" of the JPA Specification), or by using the `setHint` method of the `Query`.

The Using Query Hints example shows how to use the `eclipselink.cache-usage` hint to indicate that the cache should not be checked when reading an `Employee` for the database.

Note: Unlike the `refresh` method of the `EntityManager`, the `eclipselink.cache-usage` hint will not cause the query result to override the current cached value.

Using Query Hints

```
public Employee findEmployeeNoCache(int empId) {
      Query q = em.createQuery("SELECT e FROM Employee e WHERE
e.id =?1");
      // force read from database
      q.setHint("eclipselink.cache-usage", "DoNotCheckCache");
```

```
        q.setParameter(1, empId);
        try {
             return (Employee)q.getSingleResult();
        }
        catch(NoResultException e) {
             return null;
        }
  }
```

If you need execute this query frequently, you should use a named query. The following named query definition incorporates the cache hint from the Using Query Hints example.

```
@NamedQuery(name="findEmployeeNoCache",
          query="SELECT e FROM Employee e WHERE e.id =:empId",
          hints={@QueryHint(name="eclipselink.cache-usage",
                       value="DoNotCheckCache")})
```

The `hints` element accepts an array of `@QueryHint` annotations (see Section 8.3 "Annotations for Queries" of the JPA Specification), allowing you to set any number of hints for a query.

For more information, see the following:

- Section 3.6 "Query API" of the JPA Specification
- Using EclipseLink JPA Query Customization Extensions
- Cache
- How to Use Oracle Hints

Using the Expression API

Information pending

How to Configure Lazy Loading

By default, the EclipseLink persistence provider will use dynamic weaving to configure all applicable mappings with lazy loading (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 and 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 the following:

- Using EclipseLink JPA Extensions for Customization and Optimization
- What You May Need to Know About EclipseLink JPA Lazy Loading
- Using EclipseLink JPA Weaving
- Configuring Indirection (Lazy Loading)

How to Configure Change Tracking

By default, the EclipseLink persistence provider will use dynamic weaving to configure all applicable mappings with attribute level change tracking.

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 and 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 the following:

- Using EclipseLink JPA Extensions for Tracking Changes
- Using EclipseLink JPA Weaving
- Configuring Change Policy

How to Configure Fetch Groups

By default, the EclipseLink persistence provider will use dynamic weaving to configure all applicable mappings to use fetch groups.

For more information, see the following:

- Using EclipseLink JPA Extensions for Customization and Optimization
- Using EclipseLink JPA Weaving
- Configuring Fetch Groups

How to Use Extensions for Caching

Information pending

What You May Need to Know About EclipseLink Caching

The EclipseLink cache is an in-memory repository that stores recently read or written objects based on class and primary key values. EclipseLink uses the cache to do the following:

- Improve performance by holding recently read or written objects and accessing them in-memory to minimize database access.
- Manage locking and isolation level.

- Manage object identity.

EclipseLink uses the following two types of cache:

- the session cache maintains objects retrieved from and written to the data source;
- the unit of work cache holds objects while they participate in transactions.

When a unit of work successfully commits to the data source, EclipseLink updates the session cache accordingly.

For more information, see Cache.

What You May Need to Know About Cache Coordination

EclipseLink provides a distributed cache coordination feature that ensures data in distributed applications remains current.

For more information, see the following:

- Cache Coordination
- Configuring Locking Policy
- Querying and the Cache
- Cache

How to Configure Cascading

Information pending

For more information, see the following:

- What You May Need to Know About Cascading Entity Manager Operations
- Mapping Relationships
- Using EclipseLink JPA Extensions for Optimistic Locking
- How to Use the @PrivateOwned Annotation

What You May Need to Know About Cascading Entity Manager Operations

Typically, you use cascading in parent-child relationships.

By default, every entity manager operation applies only to the entity that you supplied as an argument to the operation. The operation will not cascade to other entities that have a relationship with the entity under operation. For some operations, such as `remove`, this is usually the desired behavior. For other operations, such as `persist`, it is not: in most cases, if you have a new entity that has a relationship to another new entity, you would

want to persist both entities together.

Using the `cascade` element of relationship annotations (see Mapping Relationships), you can define whether or not to cascade operations across relationships.

When listed as a part of the `cascade` element, you can identify the entity manager operations with the following constant values using the `javax.persitence.CascadeType` enumerated type:

- `PERSIST`–corresponds to the entity manager `persist` operation;
- `REFRESH`–corresponds to the entity manager `refresh` operation;
- `REMOVE`–corresponds to the entity manager `remove` operation;
- `MERGE`–corresponds to the entity manager `merge` operation;
- `ALL`–indicates that all four operations should be cascaded.

Note: Cascade sessions are unidirectional: you must set them on both sides of a relationship if you plan for the same behavior for both situations.

For more information, see the following:

- Mapping Relationships
- How to Configure Cascading
- How to Use the @OptimisticLocking Annotation
- How to Use the @PrivateOwned Annotation

How to Use EclipseLink Metadata

Information pending

Using EclipseLink Project

Information pending

Using sessions.xml File

Information pending

How to Use Events and Listeners

Information pending

<org.eclipse.persistence.sessions.SessionEventListener (eclipselink.session.event-listener)>

<Configure a descriptor event listener to be added during bootstrap.>

Using Session Events

Information pending

Using an Exception Handler

Information pending

What You May Need to Know About Database Platforms

EclipseLink interacts with databases using SQL. The type of database platform you choose determines the specific means by which the EclipseLink runtime accesses the database.

For more information, see Database Platforms.

What You May Need to Know About Server Platforms

You deploy your application to a specific Java EE application server.

EclipseLink supports most versions of WebLogic, OC4J, SunAS, and WebSphere application servers.

For more information, see the following:

- Configuring the Server Platform
- Integrating EclipseLink with an Application Server

How to Optimize a JPA Application

Information pending

Using Statement Caching

Information pending

Using Batch Reading and Writing

Information pending

How to Perform Diagnostics

Information pending

Using Logging

Information pending

Using Profiling

Information pending

Using JMX

Information pending

Related Topics

Packaging an EclipseLink JPA Application

Packaging means assembling all parts of the application in a way that can be correctly interpreted and used by the infrastructure when the application is deployed into an application server or run in a stand-alone JVM.

Once you chose a packaging strategy, place the `persistence.xml` file in the `META-INF` directory of the archive of your choice.

In a Java EE environment, the most efficient way to package your application is to use a tool, such as JDeveloper or Eclipse. Using OC4J, it is possible to skip the packaging step and deploy from your working directories using expanded deployment.

To package your EclipseLink JPA application, you need to 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.

For more information, see the following:

- Chapter 6 "Entity Packaging" of the JPA Specification
- Packaging an EclipseLink Application

How to Specify the Persistence Unit Name

If you are developing your application in a Java 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. The following example shows how to define the name of the persistence unit:

```
<persistence-unit name="EmployeeService">
```

For more information, see Section 6.2.1.1 "name" of the JPA Specification.

How to Specify the Transaction Type, Persistence Provider and Data Source

If you are developing your application in a Java EE environment, accept the default transaction type (see Section 6.2.1.2 "transaction-type" of the JPA Specification)–JTA (see JTA Transaction Management), and for the persistence provider setting, set the persistence provider in a `provider` element (see Section 6.2.1.2 "provider" of the JPA Specification). Specify the data source in a `jta-data-source` element, as the following example shows:

```
<persistence-unit name="EmployeeService">
      <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
 </persistence-unit>
```

Typically, you would use the JNDI access to the data source. Make this data source

available by configuring it in a server-specific configuration file or management console.

For more information, see the following sections of the JPA Specification:

- Section 6.2.1.2 "transaction-type"
- Section 6.2.1.4 "provider"
- Section 6.2.1.5 "jta-data-source, non-jta-data-source"

How to Specify 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, because 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, as the following example shows:

```
<persistence-unit name="EmployeeService">
      <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
      <mapping-file>META-
INF/employee_service_queries.xml</mapping-file>
 </persistence-unit>
```

Note that the `orm.xml` file is not listed in the previous example, because the persistence provider finds it by default.

For more information, see the following:

- Section 6.2.1.6 "mapping-file, jar-file, class, exclude-unlisted-classes" of the JPA Specification

How to Specify 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 use XML mapping).

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

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

- Local classes: the classes annotated with `@Entity` (see Section 8.1 "Entity" of the JPA Specification), `@MappedSuperclass` or `@Embeddable` in the deployment unit in which its `persistence.xml` file was packaged.

EclipseLink persistence provider, but the application server itself will discover local classes. In the Java SE environment, you can use the `exclude-unlisted-classes` element (see Section 6.2.1.6 "mapping-file, jar-file, class, exclude-unlisted-classes" of the JPA Specification) to enable this functionality–EclipseLink persistence provider will attempt to find local classes if you set this element to `false`.

- Classes in mapping files: the classes that have mapping entries, such as `entity` (see Section 10.1.2.10 "entity" of the JPA Specification), `mapped-superclass` (see Section 10.1.2.11 "mapped-superclass" of the JPA Specification) or `embeddable` (see Section 10.1.2.12 "embeddable" of the JPA Specification), in an XML mapping file.
  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 Java EE environment, you have to use the `exclude-unlisted-classes` element (see Section 6.2.1.6 "mapping-file, jar-file, class, exclude-unlisted-classes" of the JPA Specification) of the `persistence.xml` file–the use of the default setting of this element prevents local classes from being added to the persistence unit;
  - 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 (see How to Perform an Application Bootstrapping).
- Additional JAR files of managed classes: the annotated classes in a named JAR file listed in a `jar-file` element (see Section 6.2.1.6 "mapping-file, jar-file, class, exclude-unlisted-classes" of the JPA Specification) in the `persistence.xml` file. 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. The following example shows the structure of the `emp.ear` EAR file:

```
emp.ear
   emp-ejb.jar
      META-INF/persistence.xml
   employee/emp-classes.jar
      examples/model/Empoyee.class
```

The following example shows the contents of the `persistence.xml` file, with the `jar-`

`file` element containing "employee/emp-classes.jar" to reference the `emp-classes.jar` in the `employee` directory in the EAR file:

```
<persitence-unit name="EmployeeService">
      <jta-data-source>jdbc/EmployeeServiceDS</jta-data-source>
      <jar-file>employee/emp-classes.jar</jar-file>
 </persitence-unit>
```

You may choose to use any one or a combination of these mechanisms to include your managed classes in the persistence unit.

For more information, see How to Deploy an Application to Generic Java EE 5 Application Servers.

How to Add Vendor Properties

The last section in the `persistence.xml` file is the properties section. The `properties` element (see Section 6.2.1.7 "properties" of the JPA Specification) gives you the chance to supply EclipseLink persistence provider-specific settings for the persistence unit.

This example shows how to add EclipseLink-specific properties.

Using EclipseLink Persistence Provider Properties

```
<persitence-unit name="EmployeeService">
      ...
      <properties>
           <property name="eclipselink.logging.level"
value="FINE"/>

           <property name="eclipselink.cache.size.default"
value="500"/>
      </properties>
 </persitence-unit>
```

For more information, see the following:

- What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties
- Using EclipseLink JPA Extensions

How to Set Up 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. Do this by adding a classpath entry to the `META-INF/MANIFEST.MF` file in the JAR or WAR file. You may specify one or more directories or JAR files, separating them by

spaces. The following example shows how the manifest file classpath entry adds the `employee/emp-classes.jar` file and the `employee/classes` directory to the classpath of the JAR file that contains the manifest file:

```
Class-Path: employee/emp-classes.jar employee/classes
```

- Place the JAR file in the library directory of the EAR file–this will make this 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. The following example shows the `application.xml` file:

```
<application ...>
   ...
   <library-directory>myDir/jars</library-directory>
</application>
```

What You May Need to Know About Persistence Unit Packaging Options

Java 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 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 Java EE modules to share or access a persistence unit. The following example shows how to package entities in a persistence archive:

```
emp.ear
   emp-persitence.jar
      META-INF/persistence.xml
      META-INF/orm.xml
      examples/model/Employee.class
      examples/model/Phone.class
      examples/model/Address.class
      examples/model/Department.class
      examples/model/Project.class
```

Once you created 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 Section 6.2 "Persistence Unit Packaging" of the JPA Specification.

What You May Need to Know 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 Section 6.2.2 "Persistence Unit Scope" of the JPA Specification.

How to Perform an Application Bootstrapping

Outside of a container, use the `createEntityManagerFactory` method of the `javax.persistence.Persistence` class to create an entity manager factory. This method accepts a `Map` of properties and the name of the persistence unit. The properties that you pass to this method are combined with those that you already specified in the `persistence.xml` file. They may be additional properties or they may override the value of a property that you specified previously.

Note: This is a convenient way to set properties obtained from a program input, such as the command line.

This example shows how to take the user name and password properties from the command line and pass them to the EclipseLink persistence provider when creating the `EntityManagerFactory`.

Using Command-Line Persistence Properties

```
public class EmployeeService {

       public static void main (String[] args) {
            Map props = new HashMap();
            props.put("eclipselink.jdbc.user", args[0]);
            props.put("eclipselink.jdbc.password", args[1]);
            EntityManagerFactory emf =
Persistence.createEntityManagerFactory("EmployeeService", props);
            ...
            emf.close();
       }
  }
```

For more information, see the following:

- Section 7.2 "Bootstrapping in Java SE Environments" of the JPA Specification
- Application-Managed Entity Manager

Deploying an EclipseLink JPA Application

Deployment is the process of getting the application into an execution environment and running it.

For more information, see the following:

- Packaging an EclipseLink JPA Application
- How to Specify Managed Classes
- Creating EclipseLink Files for Deployment
- Deploying an EclipseLink Application
- Chapter 7 "Container and Provider Contracts for Deployment and Bootstrapping" of the JPA Specification

How to Deploy an Application to OC4J

After packaging, you deploy your EclipseLink JPA application to OC4J to execute it and make it available to end users.

You can deploy from a Java EE development tool such as JDeveloper or Eclipse.

How to Deploy an Application to Generic Java EE 5 Application Servers

Each persistence unit deployed into a Java EE container consists of a single `persistence.xml` file, any number of mapping files, and any number of class files.

Note: If you are deploying to JBoss 4.2 server, refer to How to Configure JPA Application Deployment to JBoss 4.2 Application Server.