EclipseLink

Java Persistence API (JPA) Extensions Reference for EclipseLink

Release 2.4

July 2012

EclipseLink Java Persistence API (JPA) Extensions Reference for EclipseLink

The initial contribution of this content was based on work copyrighted by Oracle and was submitted with permission.

Print date: August 2, 2012

Contents

3 Java Persistence Query Language Extensions

4 JPA Query Customization Extensions

5 eclipselink-orm.xml Schema Reference

Preface

EclipseLink provides specific annotations (*EclipseLink extensions*) in addition to supporting the standard Java Persistence Architecture (JPA) annotations. You can use these EclipseLink extensions to take advantage of EclipseLink's extended functionality and features within your JPA entities.

Audience

This document is intended for application developers who want to develop applications using EclipseLink with Java Persistence Architecture (JPA). This document does not include details about related common tasks, but focuses on EclipseLink functionality.

Developers should be familiar with the concepts and programming practices of

- Java SE and Java EE.
- Java Persistence Architecture 2.0 specification (`http://jcp.org/en/jsr/detail?id=317`)
- Eclipse IDE (`http://www.eclipse.org`)

Conventions

The following text conventions are used in this document:

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

1 Introduction

EclipseLink is an advanced, object-persistence and object-transformation framework that provides development tools and run-time capabilities that reduce development and maintenance efforts, and increase enterprise application functionality. This chapter includes the following topics:

About EclipseLink

Using EclipseLinkEclipseLink, you can integrate persistence and object-transformation into your application, while staying focused on your primary domain problem by taking advantage of an efficient, flexible, and field-proven solution.

EclipseLink is suitable for use with a wide range of Java Enterprise Edition (Java EE) and Java application architectures. Use EclipseLink to design, implement, deploy, and optimize an advanced object-persistence and object-transformation layer that supports a variety of data sources and formats, including the following: *assembly archives*, and manages deployment aspects of the software system contained therein. You create assembly archives in Oracle Virtual Assembly Builder Studio.

- Relational–for transactional persistence of Java objects to a relational database accessed using Java Database Connectivity (JDBC) drivers.
- Object-Relational Data Type–for transactional persistence of Java objects to special-purpose structured data source representations optimized for storage in object-relational data type databases such as Oracle Database.
- Enterprise information system (EIS)–for transactional persistence of Java objects to a non-relational data source accessed using a Java EE Connector architecture (JCA) adapter and any supported EIS record type, including indexed, mapped, or XML.
- XML–for non-transactional, non-prescription (in-memory) conversion between Java objects and XML Schema Document (XSD)-based XML documents using Java Architecture for XML Binding (JAXB).

EclipseLink includes support for EJB 3.0 and the Java Persistence API (JPA) in Java EE and Java SE environments including integration with a variety of application servers including:

- Oracle WebLogic Server
- Glassfish
- JBoss

- IBM WebSphere application server
- SAP NetWeaver
- Oracle OC4J
- various web containers (Apache Tomcat, IBM WebSphere CE, SpringSource tcServer)

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

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

About This Documentation

EclipseLink is the reference implementation of the Java Persistence Architecture (JPA) 2.0 specification It also includes many enhancements and extensions.

This document explains the EclipseLink enhancements and extensions to JPA. Please refer to the JPA specification for full documentation of core JPA. Where appropriate, this documentation provides links to the pertinent section of the specification.

Other Resources

For more information, see:

- Java Persistence specification for complete information about JPA
  http://jcp.org/en/jsr/detail?id=317
- EclipseLink User Guide for more information about EclipseLink support of JPA.
  http://www.eclipse.org/eclipselink/documentation/
- The EclipseLink API reference documentation (Javadoc) for complete information on core JPA plus the EclipseLink enhancements
  http://www.eclipse.org/eclipselink/api/
  - The schema for the JPA persistence configuration file
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
  - The schema for the persistence object/relational mapping file
    http://java.sun.com/xml/ns/persistence/orm_2_0.xsd
  - The schema for the native EclipseLink mapping file
    http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_0.xsd
- Examples that display the use of a number of EclipseLink JPA features
  http://wiki.eclipse.org/EclipseLink/Examples/
- JavaEE and JPA tutorial. Although this tutorial does not include EclipseLink-specific information, it does contain useful information to help you implement JPA 2.0 applications.
  http://download.oracle.com/javaee/5/tutorial/doc/bnbpy.html
- Java Persistence, a wiki-based "open book" about JPA 2.0
  http://en.wikibooks.org/wiki/Java_Persistence

2

Annotation Extensions Reference

EclipseLink supports the Java Persistence API (JPA) 2.0 specification. It also includes many enhancements and extensions. This chapter includes information on the EclipseLink extensions to the JPA annotations.

This chapter contains the following sections:

Functional Listing of Annotation Extensions

The following lists the EclipseLink annotation extensions, categorized by function:

Mapping Annotations

EclipseLink includes the following annotation extensions for mappings:

Entity Annotations

EclipseLink includes the following annotation extensions for entities:

- @AdditionalCriteria
- @ExcludeDefaultMappings
- @Multitenant
- @ReadOnly
- @OptimisticLocking
- @TenantDiscriminatorColumns
- @TenantDiscriminatorColumn
- @TenantTableDiscriminator
- @Struct

Converter Annotations

EclipseLink includes the following annotation extensions for converting data:

- @Convert
- @Converter
- @Converters
- @TypeConverter
- @TypeConverters
- @ObjectTypeConverter
- @ObjectTypeConverters
- @StructConverter
- @StructConverters

Caching Annotations

EclipseLink includes the following annotation extensions for caching:

- @Cache
- @CacheIndex
- @CacheIndexes
- @CacheInterceptor
- @TimeOfDay
- @ExistenceChecking

Customization and Optimization Annotations

EclipseLink includes the following annotation extensions for customization and optimization.

- @Customizer
- @ChangeTracking

Copy Policy Annotations

EclipseLink includes the following annotation extensions for copy policies:

- @CloneCopyPolicy
- @CopyPolicy
- @InstantiationCopyPolicy

Returning Policy Annotations

EclipseLink includes the following annotation extensions for returning policies:

- @ReturnInsert
- @ReturnUpdate

Stored Procedure and Function Annotations

EclipseLink includes the following annotation extensions for stored procedures and stored functions:

- @NamedStoredFunctionQueries
- @NamedStoredFunctionQuery
- @NamedStoredProcedureQueries
- @NamedStoredProcedureQuery
- @StoredProcedureParameter

Partitioning Annotations

EclipseLink includes the following annotation extensions for using partitions:

- @HashPartitioning
- @Partitioned
- @Partitioning
- @PinnedPartitioning
- @RangePartition
- @RangePartitioning
- @ReplicationPartitioning
- @RoundRobinPartitioning
- @UnionPartitioning
- @ValuePartitioning

Alphabetical Listing of Annotation Extensions

The following lists the EclipseLink annotation extensions:

- @AdditionalCriteria
- @Array
- @BatchFetch

- @Cache
- @CacheIndex
- @CacheIndexes
- @CacheInterceptor
- @CascadeOnDelete
- @ChangeTracking
- @ClassExtractor
- @CloneCopyPolicy
- @CompositeMember
- @ConversionValue
- @Convert
- @Converter
- @Converters
- @CopyPolicy
- @Customizer
- @DeleteAll
- @DiscriminatorClass
- @ExcludeDefaultMappings
- @ExistenceChecking
- @FetchAttribute
- @FetchGroup
- @FetchGroups
- @Field
- @HashPartitioning
- @Index
- @Indexes
- @InstantiationCopyPolicy
- @JoinFetch
- @JoinField
- @JoinFields
- @MapKeyConvert
- @Multitenant
- @Mutable
- @NamedStoredFunctionQueries
- @NamedStoredFunctionQuery
- @NamedStoredProcedureQueries
- @NamedStoredProcedureQuery

- @Noncacheable
- @NoSql
- @ObjectTypeConverter
- @ObjectTypeConverters
- @OptimisticLocking
- @OrderCorrection
- @Partitioned
- @Partitioning
- @PinnedPartitioning
- @PrimaryKey
- @PrivateOwned
- @Properties
- @Property
- @QueryRedirectors
- @RangePartition
- @RangePartitioning
- @ReadOnly
- @ReadTransformer
- @ReplicationPartitioning
- @ReturnInsert
- @ReturnUpdate
- @RoundRobinPartitioning
- @StoredProcedureParameter
- @Struct
- @StructConverter
- @StructConverters
- @Structure
- @TenantDiscriminatorColumns
- @TenantDiscriminatorColumn
- @TenantTableDiscriminator
- @TimeOfDay
- @Transformation
- @TypeConverter
- @TypeConverters
- @ValuePartition
- @UuidGenerator
- @UnionPartitioning

- @ValuePartitioning
- @VariableOneToOne
- @VirtualAccessMethods
- @WriteTransformer
- @WriteTransformers

@AdditionalCriteria

Use `@AdditionalCriteria` to define parameterized views on data.

You can define additional criteria on entities or mapped superclass. When specified at the mapped superclass level, the additional criteria definition applies to all inheriting entities, unless those entities define their own additional criteria, in which case those defined for the mapped superclass are ignored.

Annotation Elements

Table 2–1 describes this annotation's elements.

Table 2–1 @AdditionalCriteria Annotation Elements

Attribute	Description	Default
`value`	(Required) The JPQL fragment to use as the additional criteria.	

Usage

Additional criteria can provide an additional filtering mechanism for queries. This filtering option, for example, allows you to use an existing additional `JOIN` expression defined for the entity or mapped superclass and allows you to pass parameters to it.

Set additional criteria parameters through properties on the entity manager factory or on the entity manager. Properties set on the entity manager override identically named properties set on the entity manager factory. Properties must be set on an entity manager before executing a query. Do not change the properties for the lifespan of the entity manager.

Note: Additional criteria are not supported with native queries.

Examples

Specify additional criteria using the `@AdditionalCriteria` annotation or the `<additional-criteria>` element. The additional criteria definition supports any valid JPQL string and must use `this` as an alias to form the additional criteria. For example:

```
@AdditionalCriteria("this.address.city IS NOT NULL")
```

Example 2–1 shows additional criteria defined for the entity `Employee` and then shows the parameters for the additional criteria set on the entity manager.

Example 2–1 Using @AdditionalCriteria Annotation

Define additional criteria on `Employee`, as follows:

```
package model;

@AdditionalCriteria("this.company=:COMPANY")
public class Employee {

  ...
}
```

Set the property on the `EntityManager`. This example returns all employees of `MyCompany`.

```
entityManager.setProperty("COMPANY", "MyCompany");
```

Example 2–2 illustrates the same example as before, but uses the `<additional-criteria>` element in the `eclipselink-orm.xml` mapping file.

Example 2–2 Using <additional-criteria> XML

```
<additional-criteria>
  <criteria>this.address.city IS NOT NULL</criteria>
</additional-criteria>
```

Uses for Additional Criteria

Uses for additional criteria include:

- Multitenancy
- Soft Delete
- Data History
- Temporal Filtering
- Shared Table

Multitenancy

In a multitenancy environment, tenants (users, clients, organizations, applications) can share database tables, but the views on the data are restricted so that tenants have access only to their own data. You can use additional criteria to configure such restrictions.

> **Note:** In most cases, you use the `@Multitenant` annotation in multitenancy environments instead, as shown on page -82.

Example 2–3 Multitenancy Example 1

The following example restricts the data for a **Billing** client, such as a billing application or billing organization:

```
@AdditionalCriteria("this.tenant = 'Billing'")
```

Example 2–4 Multitenancy Example 2

The following example could be used in an application used by multiple tenants at the same time. The additional criteria is defined as:

```
@AdditionalCriteria("this.tenant = :tenant")
```

When the tenant acquires its `EntityManagerFactory` or `EntityManager`, the persistence/entity manager property tenant is set to the name of the tenant acquiring it. For example,

```
Map properties = new HashMap();
properties.put("tenant", "ACME");
EntityManagerFactory emf = Persistence.createEntityManagerFactory(properties);
```

Or

```
Map properties = new HashMap();
properties.put("tenant", "ACME");
EntityManager em = factory.createEntityManager(properties);
```

Soft Delete

The following example filters data that is marked as deleted (but which still exists in the table) from a query:

```
@AdditionalCriteria("this.isDeleted = false")
```

Data History

The following example returns the current data from a query, thus filtering out any out-of-date data, for example data stored in a history table.

```
@AdditionalCriteria("this.endDate is null")
```

Note: EclipseLink also provides specific history support, via `HistoryPolicy` . See Tracking Changes Using History Policy at http://wiki.eclipse.org/EclipseLink/Examples/JPA/History.

Temporal Filtering

The following example filters on a specific date:

```
@AdditionalCriteria("this.startDate <= :viewDate and this.endDate >= :viewDate")
```

Shared Table

For a shared table, there may be inheritance in the table but not in the object model. For example, a `SavingsAccount` class may be mapped to an `ACCOUNT` table, but the `ACCOUNT` table contains both savings account data (`SAVINGS`) and checking account (`CHECKING`) data. You can use additional criteria to filter out the checking account data.

See Also

For more information, see:

- "COLUMN" on page -6
- "@Multitenant" on page -82

@Array

Use `@Array` to define object-relational data types supported by specific databases, such as Oracle `VARRAY` types or PostgreSQL `JDBC Struct` types.

Annotation Elements

Table 2–2 describes this annotation's elements.

Table 2–2 @Array Annotation Elements

Annotation Element	Description	Default
`databaseType`	(Required) The name of the database array structure type.	
`targetClass`	(Optional only if the collection field or property is defined using Java generics; otherwise Required) The class (basic or embeddable) that is the element type of the collection.	Parameterized type of the collection.

Usage

Use `@Array` on a collection attribute that is persisted to an `Array` type. The collection can be of basic types or embeddable class mapped using a `Struct`.

Examples

Example 2–5 shows how to use this annotation with an Oracle `VARRAY` type.

Example 2–5 Using @Array with Oracle VARRAY

Example 2–6 shows how to use this annotation with an Postgress `Struct` type.

Example 2–6 Using @Array with Postgress Struct

See Also

For more information, see the following:

- "@Struct" on page -148

@BatchFetch

Use `@BatchFetch` to read objects related to a relationship mapping (such as one-to-one, one-to-many, many-to-one, many-to-many, and element collection) to be read in a single query.

Annotation Elements

Table 2–3 describes this annotation's elements.

Table 2–3 @BatchFetch Annotation Elements

Annotation Element	Description	Default
size	Default size of the batch fetch, used only when `BatchFetchType=IN` to definite number of keys in each `IN` clause	`256` or the query's `pageSize` (for cursor queries)
`BatchFetchType`	(optional) The type of batch fetch to use: ■ `JOIN` – The original query's selection criteria is joined with the batch query ■ `EXISTS` – Uses an SQL `EXISTS` clause and a sub-select in the batch query instead of a `JOIN` ■ `IN` – Uses an SQL `IN` clause in the batch query, passing in the source object IDs.	`JOIN`

Usage

Batch fetching allows for the optimal loading of a tree. Setting the `@BatchFetch` annotation on a *child* relationship of a tree structure causes EclipseLink to use a *single* SQL statement for each level.

Using `BatchFetchType=EXISTS` does not require an SQL `DISTINCT` statement (which may cause issues with LOBs) and may be more efficient for some types of queries or on specific databases.

When using `BatchFetchType=IN`, EclipseLink selects only objects not already in the cache. This method may work better with cursors or in situations in which you cannot use a `JOIN`. On some databases, this may only work for singleton IDs.

Examples

The following examples show how to use this annotation (and XML) with different batch fetch types.

Example 2–7 Using JOIN BatchFetch Type

```
@BatchFetch(BatchFetchType.JOIN)
@ElementCollection()
public Map<String, String> getStringMap() {
return stringMap;
}
```

```
<element-collection name="StringMap">
<batch-fetch type="JOIN">
</element-collection>
```

Example 2–8 Using EXISTS BatchFetch Type

```
@BatchFetch(BatchFetchType.JOEXISTS)
@ElementCollection()
public Map<String, String> getStringMap() {
return stringMap;
}
```

```
<element-collection name="StringMap">
<batch-fetch type="EXISTS">
</element-collection>
```

Example 2–9 Using IN BatchFetch Type

```
@BatchFetch(BatchFetchType.IN, size=50)
@ElementCollection()
public Map<String, String> getStringMap() {
return stringMap;
}
```

```
<element-collection name="StringMap">
<batch-fetch type="IN" size="50">
</element-collection>
```

See Also

For more information, see:

- "@JoinFetch" on page -74

@Cache

Use `@Cache` to configure the EclipseLink object cache. By default, EclipseLink uses a shared object cache to cache all objects. You can configure the caching type and options on a per class basis to allow optimal caching.

Annotation Elements

Table 2–4 describes this annotation's elements.

Table 2–4 @Cache Annotation Elements

Annotation Element	Description	Default
`type`	(Optional) Set this attribute to the type (`org.eclipse.persistence.annotations.CacheType` enumerated type) of the cache that you will be using: ■ `FULL` ■ `WEAK` ■ `SOFT` ■ `SOFT_WEAK` ■ `HARD_WEAK` ■ `CACHE` (not recommended) ■ `NONE` (not recommended, use `shared=false` instead) You can override this attribute with these persistence unit properties: ■ `eclipselink.cache.type.<ENTITY>` ■ `eclipselink.cache.type.default`	`CacheType.SOFT_WEAK`
`size`	(Optional) Set this attribute to an int value to define the size of cache to use (number of objects).	`100`
`shared`	(Optional) Indicate whether cached instances should be in the shared cache or in a client isolated cache. This allows the shared cache (L2 cache) to be disabled. ■ `true` —Use shared cache for cached instances. ■ `false`—Use client isolated cache for cached instances (no L2 cache).	`true`
`expiry`	(Optional) The `int` value to enable the expiration of the cached instance after a fixed period of time (milliseconds). Queries executed against the cache after this will be forced back to the database for a refreshed copy.	`.-1` (no expiry)
`expiryTimeOfDay`	(Optional) Specific time of day (`org.eclipse.persistence.annotations.TimeOfDay`) when the cached instance will expire. Queries executed against the cache after this will be forced back to the database for a refreshed copy.	`@TimeOfDay(specified=false)`
`alwaysRefresh`	(Optional) Set to a boolean value of true to force all queries that go to the database to always refresh the cache	`false`

Table 2–4 (Cont.) @Cache Annotation Elements

Annotation Element	Description	Default
`refreshOnlyIfNewer`	(Optional) Set to a boolean value of `true` to force all queries that go to the database to refresh the cache only if the data received from the database by a query is newer than the data in the cache (as determined by the optimistic locking field). **Note:** ■ This option only applies if one of the other refreshing options, such as `alwaysRefresh`, is already enabled. ■ A version field is necessary to apply this feature.	`false`
`disableHits`	(Optional) Set to a boolean value of true to force all queries to bypass the cache for hits, but still resolve against the cache for identity. This forces all queries to hit the database.	`false`
`coordinationType`	(Optional) Set this attribute to the cache coordination mode (`org.eclipse.persistence.annotations.CacheCoordinationType` enumerated type).	`CacheCoordinationType.SEND_OBJECT_CHANGES`

Usage

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

- `@Entity`
- `@MappedSuperclass`
- the root of the inheritance hierarchy (if applicable)

If you define the `@Cache` annotation on an inheritance subclass, the annotation will be ignored. If you define the `@Cache` annotation on `@Embeddable` EclipseLink will throw an exception.

Caching in EclipseLink

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:

- "Caching Overview" `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic JPA Development/Caching/Caching Overview`
- `http://wiki.eclipse.org/Introduction_to_Cache_%28ELUG%29`

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. These properties may compliment or provide an alternative to the usage of annotations.

For more information, see the following:

- How to Use the Persistence Unit Properties for Caching
  http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_Persistence_Unit_Properties_for_Caching
- What You May Need to Know About Overriding Annotations in JPA
  http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#What_You_May_Need_to_Know_About_Overriding_Annotations_in_JPA
- What You May Need to Know About Using EclipseLink JPA Persistence Unit Properties
  http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#What_You_May_Need_to_Know_About_Using_EclipseLink_JPA_Persistence_Unit_Properties

Examples

Example 2–10 illustrates an `@Cache` annotation.

Example 2–10 Using @Cache Annotation

```
...
@Entity
@Cache(
  type=CacheType.SOFT, // Cache everything until the JVM decides memory is low.
  size=64000  // Use 64,000 as the initial cache size.
  expiry=36000000,  // 10 minutes
  coordinationType=CacheCoordinationType.INVALIDATE_CHANGED_OBJECTS  // if cache
coordination is used, only send invalidation messages.
)
public class Employee {
  ...
}
```

Example 2–11 shows how to use this annotation in the `eclipselink-orm.xml` file.

Example 2–11 Using <cache> XML

```
<?xml version="1.0"?>
<entity-mappings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
  http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd"
  version="2.4">
    <entity name="Employee" class="org.acme.Employee" access="FIELD">
      <cache type="SOFT" size="64000" expiry="36000000"
coordination-type="INVALIDATE_CHANGED_OBJECTS"/>
    </entity>
</entity-mappings>
```

You can also specify caching properties at the persistence unit level (in the `persistence.xml` file) as shown here:

Example 2–12 Specifying Caching in persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_2_0.xsd"
  version="2.0">
    <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
          <property name="eclipselink.cache.shared.default" value="false"/>
          <property name="eclipselink.cache.shared.Employee" value="true"/>
          <property name="eclipselink.cache.type.Employee" value="SOFT"/>
          <property name="eclipselink.cache.size.Employee" value="64000"/>
        </properties>
    </persistence-unit>
</persistence>
```

See Also

For more information, see:

- "@ExistenceChecking" on page -56
- "@TimeOfDay" on page -166
- "@CacheInterceptor" on page -24
- "Configuring Caching" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic JPA Development/Caching/Cache Annotation

@CacheIndex

Use `@CacheIndex` to define a cached index. Cache indexes are used only when caching is enabled.

Annotation Elements

Table 2–5 describes this annotation's elements.

Table 2–5 @CacheIndex Annotation Elements

Annotation Element	Description	Default
`columnNames`	(Optional) The set of columns on which to define the index. Not required when annotated on a field/method.	
`updateable`	(Optional) Specify if the indexed field is updateable. If `true`, the object will be re-indexed on each update or refresh.	true

Usage

A cache index allows `singleResult` queries to obtain a cache hit when querying on the indexed fields. A `resultList` query cannot obtain cache hits, as it is unknown if all of the objects are in memory, (unless the cache usage query hint is used).

The index should be unique. If it is not, the first indexed object will be returned.

You can use `@CacheIndex` on an Entity class or on an attribute. The column is defaulted when defined on a attribute.

Examples

Example 2–13 shows an example of using the `@CacheIndex` annotation.

Example 2–13 Using @CacheIndex Annotation

```
@Entity
@CacheIndex(columnNames={"F_NAME", "L_NAME"}, updateable=true)
public class Employee {
  @Id
  private long id;
  @CacheIndex
  private String ssn;
  @Column(name="F_NAME")
  private String firstName;
  @Column(name="L_NAME")
  private String lastName;
}
```

Example 2–14 shows an example of using the `<cache-index>` XML element in the `eclipselink-orm.xml` file.

Example 2–14 Using <cache-index> XML

```
<?xml version="1.0"?>
<entity-mappings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd"
  version="2.4">
    <entity name="Employee" class="org.acme.Employee" access="FIELD">
        <cache-index updateable="true">
            <column-name>F_NAME</column-name>
            <column-name>L_NAME</column-name>
        </cache-index>
        <attributes>
            <id name="id"/>
            <basic name="ssn">
                <cache-index/>
            </basic>
            <basic name="firstName">
                <column name="F_NAME"/>
            </basic>
            <basic name="lastName">
                <column name="L_NAME"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

Example 2–15 shows an example query using a cache index.

Example 2–15 Caching an Index Query

```
Query query = em.createQuery("Select e from Employee e where e.firstName =
:firstName and e.lastName = :lastName");
query.setParameter("firstName", "Bob");
query.setParameter("lastName", "Smith");
Employee employee = (Employee)query.getSingleResult();
```

See Also

For more information, see:

- "@Cache" on page -16
- "Cache Indexes" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Caching/Indexes

@CacheIndexes

Use `@CacheIndexes` to define a set of `@CacheIndex` on an entity.

Annotation Elements

Table 2–6 describes this annotation's elements.

Table 2–6 @CacheIndexes Annotation Elements

Annotation Element	Description	Default
`CacheIndex[]`	An array of cache indexes	

Examples

See "@CacheIndex" on page -20 for examples of using the `@CacheIndexes` annotation.

See Also

For more information, see:

- "@CacheIndex" on page -20

@CacheInterceptor

Use `@CacheInterceptor` on an entity to intercept all EclipseLink cache access to the entity instead of responding to cache operations through an event.

Annotation Elements

Table 2–7 describes this annotation's elements.

Table 2–7 @CacheInterceptor Annotation Elements

Annotation Element	Description	Default
value	The class to be used to intercept EclipseLink's cache access.	

Usage

Once set, the specified class will receive all caching calls. Existing EclipseLink cache settings will continue to be used, any calls allowed to continue to the EclipseLink cache will execute against the configured cache.

When using with an entity in inheritance, you should define the `@CacheInterceptor` on the *root* of the inheritance hierarchy.

Examples

Example 2–16 shows how to integrate an external cache with EclipseLink.

Example 2–16 Using @CacheInterceptor Annotation

In this example, the `Employee` class intercepts all EclipseLink calls to the internal EclipseLink cache and redirects them to the Oracle Coherence Grid cache (`CoherenceInterceptor`).

```
import oracle.eclipselink.coherence.integrated.cache.CoherenceInterceptor;
import org.eclipse.persistence.annotations.Customizer;

@Entity
@CacheInterceptor(value = CoherenceInterceptor.class)
public class Employee {
...
}
```

Example 2–17 shows an example of using the `<cache-interceptor>` XML element in the `eclipselink-orm.xml` file.

Example 2–17 Using <cache-interceptor> XML

```
<entity class="Employee">
<cache-interceptor class="CoherenceInterceptor"/>
...
</entity>
```

See Also

For more information, see:

- "@Cache" on page -16

@CascadeOnDelete

Use the `@CascadeOnDelete` annotation to specify that a delete operation performed on a database object is cascaded on secondary or related tables.

`ON DELETE CASCADE` is a database foreign key constraint option that automatically removes the dependent rows.

Annotation Elements

There are no elements for this annotation.

Usage

You can place `@CascadeOnDelete` on any relationship in which the target is defined as foreign key to the source Entity.

Add the annotation on the source relationship: `@OneToOne`, `@OneToMany`, `@ManyToMany`, and `@ElementCollection` You can also add `@CascadeOnDelete` to an Entity with a `@SecondaryTable` or `JOINED` inheritance. Table 2–8 describes the affect of placing `@CascadeDelete` on these different elements

Table 2–8 Using @Cascade on Different Elements

Element	Effect of @CascadeOnDelete
Entity	Defines that secondary or joined inheritance tables should cascade the delete on the database
OneToOne mapping	The deletion of the related object is cascaded on the database. This is only allowed for mappedBy/target-foreign key OneToOne mappings (because of constraint direction).
OneToMany mapping	For a OneToMany using a `mappedBy` or `JoinColumn`, the deletion of the related objects is cascaded on the database. For a OneToMany using a `JoinTable`, the deletion of the join table is cascaded on the database (target objects cannot be cascaded even if private because of constraint direction).
ManyToMany mapping	The deletion of the join table is cascaded on the database (target objects cannot be cascaded even if private because of constraint direction).
ElementCollection mapping	The deletion of the collection table is cascaded on the database.

`@CascadeOnDelete` has the following behavior:

- DDL generation: If DDL generation is used, the generated constraint will include the cascade deletion option.
- Entity: Remove will not execute SQL for deletion from secondary or joined inheritance tables (as constraint will handle deletion).
- OneToOne: If the mapping uses cascading or orphanRemoval, SQL will not be executed to delete target object.
- OneToMany: If the mapping uses cascading or orphanRemoval, SQL will not be executed to delete target objects.
- ManyToMany: SQL will not be executed to delete from the join table.

- ElementCollection: SQL will not be executed to delete from the collection table.
- Cache: Cascaded objects will still be removed from the cache and persistence context.
- Version locking: Version will not be verified on deletion of cascaded object.
- Events: Deletion events may not be executed on the cascaded objects if the objects are not loaded.
- Cascading: The remove operation should still be configured to cascade in the mapping if using CascadeOnDelete.

Examples

Example 2–18 shows the cascading deletion of the Employee secondary table and all of its owned relationships.

Example 2–18 Using @CascadeOnDelete Annotation

```
@Entity
@SecondaryTable(name="EMP_SALARY")
@CascadeOnDelete
public class Employee{
    @Id
    private long id;
    private String firstName;
    private String lastName;
    @Column(table="EMP_SALARY")
    private String salary;
    @OneToOne(mappedBy="owner", orphanRemoval=true, cascade={CascadeType.ALL})
    @CascadeOnDelete
    private Address address;
    @OneToMany(mappedBy="owner", orphanRemoval=true, cascade={CascadeType.ALL})
    @CascadeOnDelete
    private List<Phone> phones;
    @ManyToMany
    @JoinTable(name="EMP_PROJ")
    @CascadeOnDelete
    private List<Project> projects;
    ...
}
```

In the `eclipselink-orm.xml` descriptor file, specify cascade on delete as shown in Example 2–19

Example 2–19 Using <cascade-on-delete> XML

```
...
<cascade-on-delete>true</cascade-on-delete>
...
```

See Also

For more information, see:

- EclipseLink example: http://wiki.eclipse.org/EclipseLink/Examples/JPA/DeleteCascade

- "@CascadeOnDelete"
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Schema_Generation/CascadeOnDelete

@ChangeTracking

Use `@ChangeTracking` to specify the `org.eclipse.persistence.descriptors.changetracking.ObjectChangePolicy`. This policy computes change sets for the EclipseLink UnitOfWork commit process and optimizes the transaction by including objects in the change set calculation that have at least one changed attribute.

Annotation Elements

Table 2–9 describes this annotation's elements.

Table 2–9 @ChangeTracking Annotation Elements

Annotation Element	Description	Default
`ChangeTrackingType`	(Optional) The change tracking policy to use: ■ `ATTRIBUTE` – Objects with changed attributes will be processed in the commit process to include any changes in the results of the commit. Unchanged objects will be ignored. ■ `OBJECT` – Changed objects will be processed in the commit process to include any changes in the results of the commit. Unchanged objects will be ignored. ■ `DEFERRED` – Defers all change detection to the UnitOfWork's change detection process. ■ `AUTO` – Does not set any change tracking policy; change tracking will be determined at runtime.	`AUTO`

Usage

Setting this option will improve unit of work commit performance for objects with few attributes or objects with many changed attributes.

> **Note:** When using change tracking, if you modify an object's field through reflection, EclipseLink *will not* detect the change. However, if you disable change tracking, EclipseLink *will* detect the change.

Examples

Example 2–20 shows how to use `@ChangeTracking` to set the unit of work's change policy.

Example 2–20 Using @ChangeTracking Annotation

```
@Target({TYPE})
@Retention(RUNTIME)
public @interface ChangeTracking {
   ChangeTrackingType value() default AUTO;
}
```

Example 2–21 shows how to use the `<change-tracking>` element in the `eclipselink-orm.xml` file.

Example 2–21 Using <change-tracking> XML

```
<entity class="Employee"
    <change-tracking type="AUTO"/>
...
</entity>
```

Example 2–22 shows how to configure change tracking in the persistence unit `persistence.xml` file or by importing a `property` map.

Example 2–22 Specifying Change Tracking in persistence.xml

Using `persistence.xml` file:

```
<property name="eclipselink.weaving.changetracking" value="false"/>
```

Using `property` map:

```
import org.eclipse.persistence.config.PersistenceUnitProperties;
propertiesMap.put(PersistenceUnitProperties.WEAVING_CHANGE_TRACKING, "false");
```

See Also

For more information, see:

- How to Use the @ChangeTracking Annotation
  http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Extensions_for_Tracking_Changes

@ClassExtractor

Use `@ClassExtractor` to define a custom class indicator in place of providing a discriminator column.

Annotation Elements

Table 2–10 describes this annotation's elements.

Table 2–10 @ClassExtractor Annotation Elements

Annotation Element	Description	Default
`java.lang.Class`	(Required) The name of the class extractor to apply to the entity's descriptor	

Usage

If you are mapping to an existing database, and the tables do not have a discriminator column you can still define inheritance using the `@ClassExtractor` annotation or `<class-extractor>` element. The class extractor takes a class that implements the `ClassExtractor` interface. An instance of this class is used to determine the class type to use for a database row. The class extractor must define a `extractClassFromRow` method that takes the database `Record` and `Session`.

If a class extractor is used with `SINGLE_TABLE` inheritance, the rows of the class type must be able to be filtered in queries. This can be accomplished by setting an `onlyInstancesExpression` or `withAllSubclassesExpression` for branch classes. These can be set to `Expression` objects using a `DescriptorCustomizer`.

Examples

Example 2–23shows an example of using `ClassExtractor` to define inheritance.

Example 2–23 Using @ClassExtractor Annotation

```
@Entity
@Table(name="MILES_ACCOUNT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@ClassExtractor(AirMilesClassExtractor.class)
@Customizer(AirMilesCustomizer.class)
public class AirMilesAccount implements Serializable {
    @Id
    private Long id;
    @Basic
    private String totalMiles;
    @Basic
    private String milesBalance;
    ...
}

@Entity
@Customizer(PreferredCustomizer.class)
public class PreferredAccount extends AirMilesAccount {
    ...
}

public class AirMilesClassExtractor implements ClassExtractor {
    public void extractClassFromRow(Record row, Session session) {
```

```
        if (row.get("TOTALMILES").lessThan(100000)) {
            return AirMilesAccount.class;
        } else {
            return PreferredAccount.class;
        }
    }
}

public class AirMilesCustomizer implements DescriptorCustomizer {
    public void customize(ClassDescriptor descriptor) {
        ExpressionBuilder account = new ExpressionBuilder();
        Expression expression = account.getField("TOTALMILES").lessThan(100000);
        descriptor.getInheritancePolicy().setOnlyInstancesExpression(expression);
    }
}

public class PreferredCustomizer implements DescriptorCustomizer {
    public void customize(ClassDescriptor descriptor) {
        ExpressionBuilder account = new ExpressionBuilder();
        Expression expression =
account.getField("TOTALMILES").greaterThanEqual(100000);
        descriptor.getInheritancePolicy().setOnlyInstancesExpression(expression);
    }
}
```

Example 2–24 shows how to use the `<class-extractor>` element in the `eclipselink-orm.xml` file.

Example 2–24 Using <class-extractor> XML

```
<entity class="AirMilesAccount">
    <table name="MILES_ACCOUNT"/>
    <inheritance strategy="SINGLE_TABLE"/>
    <class-extractor class="AirMilesClassExtractor"/>
...
</entity>

<entity class="PreferredAccount">
    <customizer class="PreferredCustomizer"/>
...
</entity>
```

See Also

For more information, see:

- "@Customizer" on page -48

@CloneCopyPolicy

Use `@CloneCopyPolicy` to specify an `org.eclipse.persistence.descriptors.copying.CloneCopyPolicy` on an Entity.

Annotation Elements

Table 2–11 describes this annotation's elements.

Table 2–11 @CloneCopyPolicy Annotation Elements

Annotation Element	Description	Default
`method`	(Optional) The method that will be used to create a clone for comparison with EclispeLink's `DeferredChangeDetectionPolicy`.	
`workingCopyMethod`	(Optional) The `workingCopyoMethod` that will be used to create a clone that will be used when registering an object in an EclipseLink `UnitOfWork`.	

Note: You must specify either a `method` or `workingCopyMenthod`.

Usage

You can specify `@CloneCopyPolicy` on an Entity, MappedSuperclass, or Embeddable class.

Examples

Example 2–25 and Example 2–26 show several examples of the `@CloneCopyPolicy` annotation and `<clone-copy-policy>` XML element, respectively.

Example 2–25 Using @CloneCopyPolicy Annotation

```
@CloneCopyPolicy(method="myClone")

@CloneCopyPolicy(method="myClone", workingCopyMethod="myWorkingCopyClone")

@CloneCopyPolicy(workingCopyMethod="myWorkingCopyClone")
```

Example 2–26 Using <clone-copy-policy> XML

```
<clone-copy-policy type="copy" method="myClone"
workingCopyMethod="myWorkingCopyClone"/>

<clone-copy-policy type="copy" workingCopyMethod="myWorkingCopyClone"/>

<clone-copy-policy type="copy" method="myClone"/>
```

See Also

For more information, see:

- How to Use the @CloneCopyPolicy Annotation
  http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40CloneCopyPolicy_Annotation

@CompositeMember

Use `@CompositeMember` to indicate that a class belongs to a composite persistence unit.

It should be used if target type is a primitive type and `@CollectionTable` designates the table that belongs to composite member persistence unit other than the source composite member persistence unit. This allows the source and target to be mapped to different databases.

Annotation Elements

Table 2–12 describes this annotation's elements.

Table 2–12 @CompositeMember Annotation Elements

Annotation Element	Description	Default
`value`	The name of a target composite member persistence unit to which element table belongs (if differs from source composite member persistence unit).	

Usage

The `@CompositeMember` annotation is ignored unless it is in a composite member persistence unit. It may be used in conjunction with `@ElementCollection` and `@CollectionTable` annotations.

Examples

You can configure the `CompositeMember` using annotations or the `eclipselink-orm.xml` file, as shown in these examples.

Example 2–27 Using @CompositeMember Annotation

```
@ElementCollection()
@CollectionTable(name = "MBR1_RESPONS", joinColumns=@JoinColumn(name="EMP_ID"))
@CompositeMember("composite-advanced-member_1")
@Column(name = "DESCRIPTION")
public Collection<String> getResponsibilities() {
    return responsibilities;
}
```

Example 2–28 Using <composite-member> XML

```
<element-collection name="responsibilities"
composite-member="xml-composite-advanced-member_3">
  <column name="DESCRIPTION"/>
  <collection-table name="XML_MBR3_RESPONS">
    <join-column name="EMP_ID"/>
  </collection-table>
</element-collection>
```

See Also

For more information, see:

@ConversionValue

Use `@ConversionValue` to specify the database and object values for an `ObjectTypeConverter`.

Annotation Elements

Table 2–13 describes this annotation's elements.

Table 2–13 @ConversionValue Annotation Elements

Annotation Element	Description	Default
`dataValue`	(Required) The database value.	
`objectValue`	(Required) The object value	

Usage

The JPA specification allows you to map an `Enum` to database columns using the `@Enumerated` annotation, when the database value is either the name of the `Enum` or its ordinal value. With EclipseLink, you can also map an `Enum` to a coded value, using a converter.

Examples

In Example 2–29, the `enum Gender(MALE, FEMALE)` is mapped to a single character in the database where M=MALE and F=FEMALE.

Example 2–29 Using @ConversionValue Annotation

```
@ObjectTypeConverter(name = "gender", objectType = Gender.class, dataType =
String.class, conversionValues = {
  @ConversionValue(objectValue = "Male", dataValue = "M"),
  @ConversionValue(objectValue = "Female", dataValue = "F") })

...

@Basic
@Convert("gender")
private Gender gender = Gender.Male;
```

Example 2–30 illustrates the same function using XML.

Example 2–30 Using <conversion-value> XML

```
<object-type-converter name="gender" object-type="model.Gender
"data-type="java.lang.String">
  <conversion-value object-value="Male" data-value="M" />
  <conversion-value object-value="Female" data-value="F" />
</object-type-converter>

...

<basic name="gender">
  <column name="GENDER" />
  <convert>gender</convert>
</basic>
```

See Also

For more information, see:

- "@ObjectTypeConverter" on page -102

@Convert

Use `@Convert` to specify that a named converter should be used with the corresponding mapped attribute.

Annotation Elements

Table 2–14 describes this annotation's elements.

Table 2–14 @Convert Annotation Elements

Annotation Element	Description	Default
`value`	(Optional) The `String` name for your converter	`none`

Usage

The `@Convert` has the following reserved names:

- **serialized** – Places the `org.eclipse.persistence.mappings.converters.SerializedObjectConverter` on the associated mapping.
- **class-instance** – Uses an `ClassInstanceConverter` on the associated mapping. When using a `ClassInstanceConverter`, the database representation is a `String` representing the Class name and the object-model representation is an instance of that class built with a no-args constructor
- **none** – Does not place a converter on the associated mapping.

Examples

Example 2–31 shows how to use the `@Convert` annotation to define the `gender` field.

Example 2–31 Using 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;
     }
     ...
 }
```

See Also

For more information, see:

- "@Converter" on page -42

- "@ObjectTypeConverter" on page -102
- "@TypeConverter" on page -170

@Converter

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

Annotation Elements

Table 2–15 describes this annotation's elements.

Table 2–15 @Converter Annotation Elements

Annotation Element	Description	Default
`name`	The `String` name for your converter, must be unique across the persistence unit	none
`converterClass`	The class of your converter. This class must implement the `org.eclipse.persistence.mappings.converters.Converter` interface.	none

Usage

Use `@Converter` to define a named converter that can be used with mappings. A converter can be defined on an entity class, method, or field. Specify a converter with the @Convert annotation on a Basic, BasicMap or BasicCollection mapping.

Using non-JPA Converter Annotations

EclipseLink provides a set of non-JPA converter annotations (in addition to the JPA default type mappings):

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

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

1. `@Convert`
2. `@Enumerated`
3. `@Lob`
4. `@Temporal`
5. Serialized (automatic)

Specify the converters on the following classes:

- `@Entity`
- `@MappedSuperclass`
- `@Embeddable`

Use the converters with the following mappings:

- `@Basic`
- `@Id`

- `@Version`
- `@ElementCollection`

An exception is thrown if a converter is specified with any other type of mapping annotation.

Examples

Example 2–32 shows how to use the `@Converter` annotation to specify a converter class for the `gender` field.

Example 2–32 Using the @Converter Annotation

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

Example 2–33 shows how to use the `<converter>` element in the `eclipselink-orm.xml` file.

Example 2–33 Using <converter> XML

```
<entity class="Employee">
...
    <attributes>
    ...
      <basic name="gender">
        <convert>genderConverter</convert>
        <converter name="genderConverter"
class="org.myorg.converters.GenderConverter"/>
      </basic>
    ...
    </attributes>
</entity>
```

See Also

For more information, see:

- "@Converters" on page -44
- "@Convert" on page -40
- "@MapKeyConvert" on page -80

@Converters

Use `@Converters` annotation to define multiple `@Converter` elements.

Annotation Elements

Table 2–16 describes this annotation's elements.

Table 2–16 @Converters Annotation Elements

Annotation Element	Description	Default
`Converter[]`	(Required) An array of converters	

Examples

See "@Converter" on page -42 for an example of this annotation.

See Also

For more information, see:

- "@Converter" on page -42

@CopyPolicy

Use `@CopyPolicy` to set an `org.eclipse.persistence.descriptors.copying.CopyPolicy` on an entity to produce a copy of the persistent element.

Annotation Elements

Table 2–17 describes this annotation's elements.

Table 2–17 @CopyPolicy Annotation Elements

Annotation Element	Description	Default
`java.lang.Class`	(Required) The class of the copy policy. The class must implement `org.eclipse.persistence.descriptors.copying.CopyPolicy`.	

Usage

You can specify `@CopyPolicy` on an Entity, MappedSuperclass, or Embeddable class.

Examples

Example 2–34 shows how to use this annotation.

Example 2–34 Using @CopyPolicy Annotation

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

Example 2–35 shows how to use the `<copy-policy>` element in the `eclipselink-orm.xml` file.

Example 2–35 Using <copy-policy> XML

```
<entity class="Employee">
  <table name="EMPLOYEE"/>
  <copy-policy class="mypackage.MyCopyPolicy"/>
...
</entity>
```

See Also

For more information, see:

- "@CloneCopyPolicy" on page -34
- "@InstantiationCopyPolicy" on page -72

@Customizer

Use `@Customizer` to specify a class that implements `org.eclipse.persistence.config.DescriptorCustomizer` and is to run against an entity's class descriptor after all metadata processing has been completed.

Annotation Elements

Table 2–18 describes this annotation's elements.

Table 2–18 @Customizer Annotation Elements

Annotation Element	Description	Default
`java.lang.Class`	(Required) The name of the descriptor customizer to apply to the entity's descriptor.	

Usage

You can specify `@Customizer` on an Entity, MappedSuperclass, or Embeddable class.

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

Examples

Example 2–36 show how to use this annotation.

Example 2–36 Using @Customizer Annotation

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

Example 2–37 show how to use the `<customizer>` element in the `eclipselink-orm.xml` file.

Example 2–37 Using <customizer> XML

```
<entity class="Employee">
  <table name="EMPLOYEE"/>
  <customizer class="mypackage.MyCustomizer"/>
...
</entity>
```

See Also

For more information, see:

- "How to Use the @Customizer Class" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40Customizer_Annotation

- EclipseLink Examples
  http://wiki.eclipse.org/EclipseLink/Examples/JPA/MappingSelectionCriteria
- "Customizers"
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Customizers

@DeleteAll

Use `@DeleteAll` to indicate that when an relationship is deleted, EclipseLink should use a delete all query. This typically happens if the relationship is `PrivateOwned` and its owner is deleted. In that case, the members of the relationship will be deleted without reading them in.

Annotation Elements

There are no elements for this annotation.

Usage

WARNING: Use this annotation with caution. EclipseLink will not validate whether the target entity is mapped in such a way as to allow the delete all to work.

Examples

Example 2–38 shows how to use `@DeleteAll` on a relationship mapping.

Example 2–38 Using @DeleteAll Annotation

```
@Entity
public class Department {
  ...
  @OneToMany(mappedBy = "department")
  @PrivateOwned
  @DeleteAll
  public List<Equipment> getEquipment() {
    return equipment;
    }
  ...
  }
```

Example 2–38 shows how to use the `<delete-all>` element in the `eclipselink-orm.xml` file.

Example 2–39 Using <delete-all> XML

```
<entity class="Department">
  ...
  <attributes>
    <one-to-many name="equipment" target-entity="Equipment"
mapped-by="department">
      <private-owned/>
      <delete-all/>
    </one-to-many>
...
</attributes>
```

```
</entity>
```

See Also

For more information, see:

- "@PrivateOwned" on page -122

@DiscriminatorClass

Use `@DiscriminatorClass` with a `@VariableOneToOne` annotation to determine which entities will be added to the list of types for the mapping.

Annotation Elements

Table 2–19 describes this annotation's elements.

Table 2–19 @DiscriminatorClass Annotation Elements

Annotation Element	Description	Default
`discriminator`	(Required) The discriminator to be stored in the database.	
`value`	(Required) The class to be instantiated with the `discriminator`.	

Usage

The `@DiscriminatorClass` annotation can be specified only within a `@VariableOneToOne` mapping.

Examples

See "@VariableOneToOne" on page -184 for an example of a variable one-to-one mapping with `@DiscriminatorClass`.

See Also

For more information, see:

- "@VariableOneToOne" on page -184

@ExcludeDefaultMappings

Use `@ExcludeDefaultMappings` to specify that no default mapping should be added to a specific class. Instead, EclipseLink will use only mappings that are explicitly defined by annotations or the XML mapping file.

Annotation Elements

There are no elements for this annotation.

Usage

You can specify `@ExcludeDefaultMappings` on an Entity, MappedSuperclass, or Embeddable class.

Examples

See Also

For more information, see:

@ExistenceChecking

Use `@ExistenceChecking` to specify how EclipseLink should check to determine determine if an entity is new or exists.

On `merge()` operations, use `@ExistenceChecking` to specify if EclipseLink uses only the cache to determine if an object exists, or if the object should be read (from the database or cache). By default the object will be read.

Annotation Elements

Table 2–20 describes this annotation's elements.

Table 2–20 @ExistenceChecking Annotation Elements

Annotation Element	Description	Default
`ExistenceType`	(Optional) Set the existence checking type: ■ `ASSUME_EXISTENCE` ■ `ASSUME_NON_EXISTENCE` ■ `CHECK_CHACHE` ■ `CHECK_DATABASE`	`CHECK_CACHE`

Usage

You can specify `@ExistenceChecking` on an Entity or MappedSuperclass.

EclipseLink supports the following existence checking types:

- `ASSUME_EXISTENCE` – If the object's primary key does not include `null` then it must exist. You may use this option if the application guarantees or does not care about the existence check.
- `ASSUME_NON_EXISTENCE` – Assume that the object does not exist. You may use this option if the application guarantees or does not care about the existence check. This will always force an `INSERT` operation.
- `CHECK_CHACHE` – If the object's primary key does not include `null` and it is in the cache, then it must exist.
- `CHECK_DATABASE` – Perform a "does exist check" on the database.

Examples

See "@Cache" on page -16 for examples of using `@TimeOfDay`.

See Also

For more information, see:

- "@Cache" on page -16
- "@TimeOfDay" on page -166

@FetchAttribute

Use `@FetchAttribute` to improve performance within a fetch group; it allows on-demand loading of a group of an object's attributes. As a result, the data for an attribute might not be loaded from the datasource until an explicit access call occurs.

This avoids loading all the data of an object's attributes if the user requires only some of the attributes.

Annotation Elements

Table 2–21 describes this annotation's elements.

Table 2–21 @FetchAttribute Annotation Elements

Annotation Element	Description	Default
`name`	(Required) Name of the fetch attribute.	

Usage

EclipseLink provides two types of fetch groups:

- Pre-defined fetch groups at the Entity or MappedSuperclass level
- Dynamic (use case) fetch groups at the query level

You should extensively review your use cases when using fetch groups. In many cases, additional round-trips will offset any gains from deferred loading.

Examples

Example 2–40 shows how to use `@FetchAttribute` within a `@FetchGroup` annotation.

Example 2–40 Using @FetchAttribute Annotation

```
@Entity
@FetchGroup(name="default-fetch-group", attributes={
        @FetchAttribute(name="id"),
        @FetchAttribute(name="name"),
        @FetchAttribute(name="address")})
@DefaultResource("default-fetch-group")
public class Person {

   @Id
   private int id;

   private String name;

   @OneToOne(fetch=LAZY)
   private Address address;

   @ManyToOne(fetch=EAGER)
   private ContactInfo contactInfo;
```

See Also

For more information, see:

- "@FetchGroup" on page -60

@FetchGroup

Use `@FetchGroup` to load a group of attributes on demand, as needed.

This avoids wasteful practice of loading all data of the object's attributes, if which the user is interested in only partial of them. However, it also means that the data for an attribute might not loaded from the underlying data source until an explicit access call for the attribute first occurs.

Annotation Elements

Table 2–22 describes this annotation's elements.

Table 2–22 @FetchGroup Annotation Elements

Annotation Element	Description	Default
`FetchAttribute[] attributes`	(Required) The list of attributes to fetch.	none
`java.lang.String name`	(Required) The fetch group name.	none
`boolean load`	(Optional) Indicates whether all relationship attributes specified in the fetch group should be loaded.	`false`

Usage

You should perform a careful use case analysis when using `@FetchGroup`; any gains realized from the deferred loading could be offset by the extra round-trip.

EclipseLink supports fetch groups at two levels:

- Pre-defined fetch groups at the Entity or MappedSuperclass level
- Dynamic (use case) fetch groups at the query level

You can use fetch groups only when using weaving or when individual classes that define them explicitly implement the `org.eclipse.persistence.queries.FetchGroupTracker interface`.

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

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

You can use fetch groups in JPA projects for EJB objects, as well as for POJO classes. For POJO classes, use partial object querying.

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

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

Use `AttributeGroup` to configure the use of partial entities in fetch, load, copy, and merge operations.

Examples

Example 2–41 show how to use this annotation.

Example 2–41 Using @FetchGroup Annotation

```
@FetchGroup(name="names", attributes={
    @FetchAttribute(name="firstName"),
    @FetchAttribute(name="lastName")})
```

Example 2–41 show how to use this feature in the `eclipselink-orm.xml` file.

Example 2–42 Using <fetch-group> XML

```
<entity class="model.Employee">
    <secondary-table name="SALARY" />
    <fetch-group name="names">
        <attribute name="firstName" />
        <attribute name="lastName" />
    </fetch-group>
...
```

You can also use a named fetch group with a query, as shown in Example 2–43.

Example 2–43 Using a Named Fetch Group on a Query

```
TypedQuery query = em.createQuery("SELECT e FROM Employee e", Employee.class);

query.setHint(QueryHints.FETCH_GROUP_NAME, "names");
```

See Also

For more information, see:

- "@FetchAttribute" on page -58
- "@FetchGroups" on page -62

@FetchGroups

Use `@FetchGroups` to define a group of `@FetchGroup`.

Annotation Elements

Table 2–23 describes this annotation's elements.

Table 2–23 @FetchGroups Annotation Elements

Annotation Element	Description	Default
`FetchGroup`	(Required) An array of fetch groups (`@FetchGroup`)	

Usage

You can specify `@FetchGroups` on an Entity or MappedSuperclass.

You can also enable or disable fetch groups through weaving for the persistence unit.

Examples

See "@FetchGroup" on page -60 for an example of using fetch groups.

Example 2–44 shows how to configure fetch groups in the persistence unit `persistence.xml` file or by importing a `property` map.

Example 2–44 Specifying Fetch Groups in persistence.xml

Using `persistence.xml` file:

```
<property name="eclipselink.weaving.fetchgroups" value="false"/>
```

Using `property` map:

```
import org.eclipse.persistence.config.PersistenceUnitProperties;
propertiesMap.put(PersistenceUnitProperties.WEAVING_FETCHGROUPS, "false");
```

See Also

For more information, see:

- "@FetchGroup" on page -60
- "@FetchAttribute" on page -58
- "Using EclipseLink JPA Weaving" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#Using_EclipseLink_JPA_Weaving

@Field

Use `@Field` to define a structured data type's field name for an object mapped to NoSql data.

Annotation Elements

Table 2–24 describes this annotation's elements.

Table 2–24 @Field Annotation Elements

Annotation Element	Description	Default
`name`	(Optional) The data type's name of the field.	

Usage

The `@Field` annotation is a generic form of the `@Column` annotation, which is not specific to relational databases. You can use `@Field` to map EIS and NoSQL data.

Examples

See "@NoSql" on page -100 for an example of the `@Field` annotation.

See Also

For more information, see:

- "@NoSql" on page -100

@HashPartitioning

Use `@HashPartitioning` to partition access to a database cluster by the hash of a field value from the object (such as the object's location or tenant). The hash indexes into the list of connection pools.

Annotation Elements

Table 2–25 describes this annotation's elements.

Table 2–25 @HashPartitioning Annotation Elements

Annotation Element	Description	Default
name	(Required) The name of the partition policy. The name must be unique within the persistence unit.	
`partitionColumn`	(Required) The database column or query parameter by which to partition queries.	
`connectionPolls`	(Optional) List of connection pool names across which to partition.	All defined pools in the ServerSession
`unionUnpartitionableQueries`	(Optional) Specify if queries that *do not* contain the partition hash should be sent to every database union the result.	False

Usage

All write or read requests for objects with the hash value are sent to the server. Queries that do not include the field as a parameter will be:

- Sent to all servers and unioned

  or

- Handled based on the session's default behavior.

You can enable partitioning on an Entity, relationship, query, or session/persistence unit. Partition policies are globally named (to allow reuse) and must set using the `@Partitioned` annotation.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

If a transaction modifies data from multiple partitions, you should use JTA ensure proper two-phase commit of the data. You can also configure an exclusive connection in the EntityManager to ensure that only a single node is used for a single transaction.

Examples

See "@Partitioned" on page -110 for an example of partitioning with EclipseLink.

See Also

For more information, see:

- "Data Partitioning"
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning

- "@Partitioned" on page -110

@Index

An index is a database structure defined for a table, to improve query and look-up performance for a set of columns. Use the `@Index` annotation in code or the `<index>` element in the `eclipselink-orm.xml` descriptor to create an index on a table.

An index can be defined on an entity or on an attribute. For the entity it must define a set of columns to index.

Index creation is database specific. Some databases may not support indexes. Most databases auto-index primary key and foreign key columns. Some databases support advanced index DDL options. To create more advanced index DDL, a DDL script or native query can be used.

Annotation Elements

Table 2–26 describes this annotation's elements.

Table 2–26 @Index Annotation Elements

Annotation Element	Description	Default
`java.lang.String catalog`	(Optional) The catalog of the `INDEX`.	Default catalog
`java.lang.String[] columnNames`	(Not required when annotated on a field or method) Specify the set of columns to define the index on.	For an Entity, the table. For an attribute, the table and column.
`java.lang.String name`	(Optional) The name of the `INDEX`.	`<table>_<column>_INDEX` (but a name should be provided)
`java.lang.String schema`	(Optional) The schema of the `INDEX`.	Default schema
`java.lang.String table`	(Optional) The table to define the index on; defaults to entities primary table.	The entity's primary table.
`boolean unique`	(Optional) Specify whether the index is unique or non-unique.	`false`

Usage

Examples

This example defines three indexes, one on **first name**, one on **last name**, and a multiple column index on **first name** *and* **last name**.

Example 2–45 Using @Index Annotation

```
@Entity
@Index(name="EMP_NAME_INDEX", columns={"F_NAME","L_NAME"})
public class Employee{
    @Id
    private long id;
    @Index
    @Column(name="F_NAME")
    private String firstName;
    @Index
    @Column(name="L_NAME")
```

```
    private String lastName;
    ...
}
```

You can also create an index in the `eclipselink-orm.xml` descriptor using `<index>`, as shown in the following example. Define columns using the `<column>` subelement. All the attributes supported in the `@Index` annotation are also supported in the `<index>` element.

Example 2–46 Using <index> XML

```
<index name="EMP_NAME_INDEX" table="EMPLOYEE" unique="true">
    <column>F_NAME</column>
    <column>L_NAME</column>
</index>
```

See Also

For more information see:

- "@Indexes" on page -70

@Indexes

Use `@Indexes` to define a set of database indexes for an Entity.

Annotation Elements

Table 2–27 describes this annotation's elements.

Table 2–27 @Indexes Annotation Elements

Annotation Element	Description	Default
`Index[]`	An array of database indexes	

Examples

See "@Index" on page -68 for an example of using the `@Index` annotation.

See Also

For more information see:

- "@CopyPolicy" on page -46
- "@CloneCopyPolicy" on page -34
- "@Index" on page -68

@InstantiationCopyPolicy

Use `@InstantiationCopyPolicy` to set an `org.eclipse.persistence.descriptors.copying.InstantiationCopyPolicy` on an Entity.

Because `InstantiationCopyPolicy` is the default EclipseLink copy policy, this annotation needed only to override other types of copy policies.

Annotation Elements

Table 2–28 describes this annotation's elements.

Table 2–28 @InstantiationCopyPolicy Annotation Elements

Annotation Element	Description	Default

Usage

You can specify `@InstantiationCopyPolicy` on an Entity, MappedSuperclass, or Embeddable entity.

Instantiation is the default copy policy in EclipseLink.

Examples

Example 2–47 Using @InstantiationCopyPolicy Annotation

Example 2–48 Using <instantiation-copy-policy> XML

See Also

For more information, see:

- "@CopyPolicy" on page -46
- "@CloneCopyPolicy" on page -34
- "How to Use the @InstantiationCopyPolicy Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40InstantiationCopyPolicy_Annotation

@JoinFetch

Use the `@JoinFetch` annotation to enable the joining and reading of the related objects in the same query as the source object.

> **Note:** You should set join fetching at the query level, as not all queries require joining.

Annotation Elements

Table 2–29 describes this annotation's elements.

Table 2–29 @JoinFetch Annotation Elements

Annotation Element	Description	Default
`value`	(Optional) 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.	`JoinFetchType.INNER`

Usage

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

- `@OneToOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`
- `@BasicCollection` (deprecated)
- `@BasicMap` (deprecated)

Alternatively, you can use batch reading, especially for collection relationships.

Examples

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

Example 2–49 Using @JoinFetch Annotation

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

```
        return managedEmployees;
    }
    ...
}
```

See Also

For more information, see:

- "@BatchFetch" on page -14

@JoinField

Use `@JoinField` to define a structured data type's foreign key field for an object mapped to NoSql data.

Annotation Elements

Table 2–30 describes this annotation's elements.

Table 2–30 @JoinField Annotation Elements

Annotation Element	Description	Default
`name`	(Optional) The name of the foreign key/ID reference field in the source record.	
`referencedFieldName`	(Optional) The name of the ID field in the target record.	

Usage

The `@JoinField` annotation is a generic form of the `@JoinColumn` annotation, which is not specific to relational databases. You can use `@JoinField` to map EIS and NoSQL data.

Examples

See Also

For more information, see:

- "Mappings" `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/NoSQL/Mappings`
- "@JoinFields" on page -78

@JoinFields

Use `@JoinFields` to define a set of `@JoinField` annotations on a relationship.

Annotation Elements

Table 2–31 describes this annotation's elements.

Table 2–31 @JoinFields Annotation Elements

Annotation Element	Description	Default
`JoinField[]`J	An array of join fields	

Examples

See "@JoinField" on page -76 for an example of using the `@Index` annotation.

See Also

For more information, see:

- "@JoinField" on page -76

@MapKeyConvert

Use `@MapKeyConvert` to specify a named converter to be used with the corresponding mapped attribute key column.

Annotation Elements

Table 2–32 describes this annotation's elements.

Table 2–32 @MapKeyConvert Annotation Elements

Annotation Element	Description	Default
`value`	(Optional) Name of the converter to use: ■ `serialized` ■ `class-instance` ■ `none` ■ custom converter	none

Usage

The `@MapKeyConvert` annotation has the following reserved names:

- `serialized`: Will use a `SerializedObjectConverter` on the associated mapping. When using a `SerializedObjectConverter` the database representation is a binary field holding a serialized version of the object and the object-model representation is a the actual object
- `class-instance`: Will use an ClassInstanceConverter on the associated mapping. When using a ClassInstanceConverter the database representation is a String representing the Class name and the object-model representation is an instance of that class built with a no-args constructor
- `none` - Will place no converter on the associated mapping. This can be used to override a situation where either another converter is defaulted or another converter is set.

If you do not use one of these reserved names, you must define a custom converter, using the `@Converter` annotation.

Examples

Example 2–50 shows using a `@MapKeyConvert` annotation to apply a converter to a map's key.

Example 2–50 Using @MapKeyConvert Annotation

```
@Entity
public class Entity
 …
    @ElementCollection
    @MapKeyColumn(name="BANK")
    @Column(name="ACCOUNT")
    @Convert("Long2String")
    @MapKeyConvert("CreditLine")
    public Map getCreditLines() {
        return creditLines;
    }
```

Example 2–51 shows how to use the <map-key-convert> element in the eclipselink-orm.xml file.

Example 2–51 Using <map-key-convert> XML

```
<element-collection name="creditLines">
  <map-key-convert>CreditLine</map-key-convert>
  <map-key-column name="BANK"/>
  <column name="ACCOUNT"/>
  <convert>Long2String</convert>
  <object-type-converter name="CreditLine">
    <conversion-value data-value="RBC" object-value="RoyalBank"/>
    <conversion-value data-value="CIBC" object-value="CanadianImperial"/>
    <conversion-value data-value="SB" object-value="Scotiabank"/>
    <conversion-value data-value="TD" object-value="TorontoDominion"/>
  </object-type-converter>
  <type-converter name="Long2String" data-type="String" object-type="Long"/>
  <collection-table name="EMP_CREDITLINES">
    <join-column name="EMP_ID"/>
  </collection-table>
</element-collection>
```

See Also

For more information, see:

- "@Converter" on page -42
- "@Convert" on page -40

@Multitenant

The `@Multitenant` annotation specifies that a given entity is shared among multiple tenants of an application. The multitenant type specifies how the data for these entities are to be stored on the database for each tenant. Multitenancy can be specified at the entity or mapped superclass level.

Annotation Elements

Table 2–33 describes this annotation's elements.

Table 2–33 @Multitenant Annotation Elements

Annotation Element	Description	Default
`boolean IncludeCriteria`	Indicates if the database requires the tenant criteria to be added to the `SELECT`, `UPDATE`, and `DELETE` queries.	`true`
`MultitenantType value`	Specifies the multitenant strategy to use: `SINGLE_TABLE`, `TABLE_PER_TENANT`, or `VDP`.	`SINGLE_TABLE`

Usage

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

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

Three types of multitenancy are available:

- Single-Table Multitenancy
- Table-Per-Tenanat Multitenancy
- VDP Multitenancy

Single-Table Multitenancy

The SINGLE_TABLE multitenant type specifies that any table to which an entity or mapped superclass maps can include rows for multiple tenants. Access to tenant-specific rows is restricted to the tenant.

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

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

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

For more information how to use tenant discriminator columns to configure single-table multitenancy, see "@TenantDiscriminatorColumn" on page 156.

Examples

The following example uses @Multitenant, @TenantDiscriminatorColumn, and a context property to define single-table multitenancy on an entity:

Example 2–52 Example Using @Multitenant

```
@Entity
@Table(name="EMP")
@Multitenant(SINGLE_TABLE)
@TenantDiscriminatorColumn(name = "TENANT_ID",
   contextProperty = "employee-tenant.id")
```

The following example uses the <multitenant> element to specify a minimal single-table multitenancy. SINGLE_TABLE is the default value and therefore does not have to be specified.

Example 2–53 Example Using <multitenant>

```
<entity class="model.Employee">
  <multitenant/>
  <table name="EMP"/>
  ...
</entity>
```

Table-Per-Tenanat Multitenancy

The `TABLE_PER_TENANT` multitenant type specifies that the table(s) (`Table` and `SecondaryTable`) for an entity are tenant-specific tables based on the tenant context.. Access to these tables is restricted to the specified tenant. Relationships within an entity that use a join or collection table are also assumed to exist within that context.

As with other multitenant types, table-per-tenant multitenancy can be specified at the entity or mapped superclass level. At the entity level, a tenant context property must be provided on each entity manager after a transaction has started.

Table-per-tenant entities can be mixed with other multitenant-type entities within the same persistence unit.

All read, insert, update, and delete operations for the tenant apply only to the tenant's table(s).

Tenants share the same server session by default. The table-per-tenant identifier must be set or updated for each entity manager. ID generation is assumed to be unique across all the tenants in a table-per-tenant strategy.

To configure table-per-tenant multitenancy, you must specify:

- A table-per-tenant property to identify the user. This can be set per entity manager, or it can be set at the entity manager factory to isolate table-per-tenant per persistence unit.)
- A tenant table discriminator to identify and isolate the tenant's tables from other tenants' tables. The discriminator types are `SCHEMA`, `SUFFIX`, and `PREFIX`. For more information about tenant discriminator types, see "@TenantTableDiscriminator" on page -164

  .

Examples

The following example shows the `@Multitenant` annotation used to define table-per-tenant multitenancy on an entity. `@TenantTableDiscriminator(SCHEMA)` specifies that the discriminator table is identified by schema.

Example 2–54 Example Using @Multitenant with @TenantTableDiscriminator

```
@Entity
@Table(name="EMP")
@Multitenant(TABLE_PER_TENANT)
@TenantTableDiscriminator(SCHEMA)
public class Employee {
    ...
}
```

The following example shows the `<multitenant>` element and the `<tenant-table-discriminator>` elements used to define a minimal table-per-tenant multitenancy.

Example 2–55 Example Using <multitenant> with <tenant-table-discriminator>

```
<entity class="Employee">
  <multitenant type="TABLE_PER_TENANT">
    <tenant-table-discriminator type="SCHEMA"/>
  </multitenant>
  <table name="EMP">
```

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

VDP Multitenancy

The `VPD` (Virtual Private Database) multitanancy type specifies that the database handles the tenant filtering on all SELECT, UPDATE and DELETE queries. To use this type, the platform used with the persistence unit must support VPD.

To use EclipseLink VPD multitenancy, you must first configure VPD in the database and then specify multitenancy on the entity or mapped superclass, using `@Multitenant` and `@TenantDiscriminatorColumn`:

Examples

Example 2–56 shows VPD multitenancy defined on an entity. As noted above, VPD in the database must also be configured to enable VPD multitenancy. In this case, the VPD database was configured to use the USER_ID column to restrict access to specified rows by specified clients. Therefore, `USER_ID` is also specified as the tenant discriminator column for the EclipseLink multitenant operations.

Example 2–56 Example Using @Multitenancy

The following example shows

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

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

The following example shows...

Example 2–57 Example Using <multitenant>

```
<entity class="model.Employee">
  <multitenant type="VPD">
    <tenant-discriminator-column name="USER_ID" context-property="tenant.id"/>
  </multitenant>
  <table name="EMPLOYEE"/>
  ...
</entity>
```

See Also

- "@TenantDiscriminatorColumn" on page 156
- "@TenantDiscriminatorColumns" on page 162
-
- Multitenant Examples at
  http://wiki.eclipse.org/EclipseLink/Examples/JPA/Multitenant

@Mutable

Use `@Mutable` on a `@Basic` mapping to specify if the value of a complex field type can be *changed* (or not changed) instead of being *replaced*. Mutable mappings may affect the performance of change tracking; attribute change tracking can only be weaved with non-mutable mappings.

Annotation Elements

Table 2–34 describes this annotation's elements.

Table 2–34 @Mutable Annotation Elements

Annotation Element	Description	Default
`boolean value`	(Optional) Specify if the mapping is mutable.	true

Usage

Most basic types (such as `int`, `long`, `float`, `double`, `String`, and `BigDecimal`) are not mutable.

By default, `Date` and `Calendar` types are assumed to be not mutable. To make these types mutable, use the `@Mutable` annotation. You can also use the global persistence property `eclipselink.temporal.mutable` to set the mappings as mutable.

By default, serialized types are assumed to be mutable. You can set the `@Mutable` annotation to `false` to make these types not mutable.

You can also configure mutable mappings for `Date` and `Calendar` fields in the persistence unit in the `persistence.xml` file.

Examples

Example 2–58 shows how to use the `@Mutable` annotation to specify `Employee` field `hireDate`.

Example 2–58 Using @Mutable Annotation

```
@Entity
public class Employee implements Serializable {

...

  @Temporal(DATE)
    @Mutable
    public Calendar getHireDate() {
    return hireDate;
  }

..
 }
```

Example 2–59 shows how to configure mutable mappings in the persistence unit `persistence.xml` file or by importing a `property` map.

Example 2–59 Specifying Mutable Mappings in persistence.xml

Using `persistence.xml` file:

```
<property name="eclipselink.temporal.mutable" value="true"/>
```

Using `property` map:

```
import org.eclipse.persistence.config.PersistenceUnitProperties;
propertiesMap.put(PersistenceUnitProperties.TEMPORAL_MUTABLE, "false");
```

See Also

For more information, see:

- "Mapping Annotations" on page 2-1

@NamedStoredFunctionQueries

Use `@NamedStoredFunctionQueries` annotation to define multiple `NamedStoredFunctionQuery` items.

Annotation Elements

Table 2–35 describes this annotation's elements.

Table 2–35 @NamedStoredFunctionQueries Annotation Elements

Annotation Element	Description	Default
`NamedStoredFunctionQuery[]`	(Required) An array of named stored procedure query.	

Examples

Example 2–60 shows how to use this annotation.

Example 2–60 Using @NamedStoredFunctionQueries Annotation

```
XML is just a list multiple named-stored-function-query elements in a row
@NamedStoredFunctionQueries{(
    @NamedStoredFunctionQuery(
    name="StoredFunction_In",
    functionName="StoredFunction_In",
    parameters={
        @StoredProcedureParameter(direction=IN, name="P_IN", queryParameter="P_
IN", type=Long.class)
    },
    returnParameter=@StoredProcedureParameter(queryParameter="RETURN",
type=Long.class))
)}
```

To define multiple named stored procedures in the `eclipselink-orm.xml` file, simply create a list of multiple `<named-stored-function_query>` elements.

See Also

For more information, see:

- "@NamedStoredFunctionQuery" on page -92

@NamedStoredFunctionQuery

Use `@NamedStoredFunctionQuery` to define queries that call stored functions as named queries.

Annotation Elements

Table 2–36 describes this annotation's elements.

Table 2–36 @NamedStoredFunctionQuery Annotation Elements

Annotation Element	Description	Default
`functionName`	(Required) The name of the stored function.	
name	(Required) The unique name that references this stored function query.	
`returnParamter`	(Required) The return value of the stored function.	
`callByIndex`	(Optional) Specifies if the stored function should be called by **index** or by **name**. ■ If by index, the parameters must be defined in the same order as the procedure on the database. ■ If by name, you must use the database platform support naming procedure parameters	false
`hints`	(Optional) Query hints	
`parameters`	(Optional) The parameters for the stored function.	
`resultSetMapping`	(Optional) The name of the SQLResultMapping.	

Usage

You can specify `@NamedStoredFunctionQuery` on an Entity or MappedSuperclass.

Examples

Example 2–61 shows how to use this annotation.

Example 2–61 Using @NamedStoredFunctionQuery Annotation

```
@Entity
@Table(name="CMP3_ADDRESS")

@NamedStoredFunctionQuery(
  name="StoredFunction_In",
  functionName="StoredFunction_In",
  parameters={
    @StoredProcedureParameter(direction=IN, name="P_IN", queryParameter="P_IN",
type=Long.class)
    },
  returnParameter=@StoredProcedureParameter(queryParameter="RETURN",
type=Long.class)
  )
public class Address implements Serializable {
...
}
```

Example 2–62 shows how to use the `<named-stored-function-query>` element in the `eclipselink-orm.xml` file.

Example 2–62 Using <named-stored-function-query> XML

```
<named-stored-function-query name="StoredFunction_In"
procedure-name="StoredFunction_In">
    <parameter direction="IN" name="P_IN" query-parameter="P_IN" type="Long"/>
</named-stored-function-query>
```

See Also

For more information, see:

- "@NamedStoredFunctionQueries" on page -90

@NamedStoredProcedureQueries

Use `@NamedStoredProcedureQueries` annotation to define multiple `NamedStoredProcedureQuery` items.

Annotation Elements

Table 2–37 describes this annotation's elements.

Table 2–37 @NamedStoredProcedureQueries Annotation Elements

Annotation Element	Description	Default
`value`	(Required) An array of named stored procedure query.	

Examples

Example 2–63 shows how to use this annotation.

Example 2–63 Using @NamedStoredProcedureQueries Annotation

```
@Entity
@Table(name="EMPLOYEE")
@NamedStoredProcedureQueries({
  @NamedStoredProcedureQuery(
    name="ReadEmployeeInOut",

resultClass=org.eclipse.persistence.testing.models.jpa.customfeatures.Employee.cla
ss,
    procedureName="Read_Employee_InOut",
    parameters={
      @StoredProcedureParameter(direction=IN_OUT, name="employee_id_v",
queryParameter="ID", type=Integer.class),
      @StoredProcedureParameter(direction=OUT, name="nchar_v",
queryParameter="NCHARTYPE", type=Character.class)}
    ),
    @NamedStoredProcedureQuery(
      name="ReadEmployeeCursor",

resultClass=org.eclipse.persistence.testing.models.jpa.customfeatures.Employee.cla
ss,
      procedureName="Read_Employee_Cursor",
      parameters={
        @StoredProcedureParameter(direction=IN, name="employee_id_v",
queryParameter="ID", type=Integer.class),
        @StoredProcedureParameter(direction=OUT_CURSOR, queryParameter="RESULT_
CURSOR")})
})
public class Employee implements Serializable {
```

To define multiple named stored procedure queries in the `eclipselink-orm.xml` file, simply create a list of multiple `<named-stored-procedure_query>` elements.

See Also

For more information, see:

- "@NamedStoredProcedureQuery" on page -96

@NamedStoredProcedureQuery

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

Annotation Elements

Table 2–38 describes this annotation's elements.

Table 2–38 @NamedStoredProcedureQuery Annotation Elements

Annotation Element	Description	Default
`name`	(Required) Unique name that references this stored procedure query.	
`procedureName`	(Required) Name of the stored procedure	
`callByIndex`	(Optional) Specifies if the stored procedure should be called by name. ■ If **true**, the StoredProcedureParameters must be defined in the same order as the procedure on the database ■ If **false**, the database platform must support naming procedure parameters	`false`
`hints`	(Optional) An array of query hints.	
`multipleResultSets`	(Optional) Specifies if the stored procedure returns multiple result sets. This applies only for databases that support multiple result sets from stored procedures.	`false`
`parameters`	(Optional) An array of parameters for the stored procedure	
`resultClass`	(Optional) The class of the result	`void.class`
`resultSetMapping`	(Optional) Name of the `SQLResultMapping`	
`returnsResultSet`	(Optional) Specifies if the stored procedure retainers a result set This applies only for databases that support result sets from stored procedures.	`false`

Usage

You can specify `@NamedStoredProcedureQuery` on an Entity or MappedSuperclass.

Examples

Example 2–64 shows how to use `@NamedStoredProcedureQuery` to define a stored procedure.

Example 2–64 Using @NamedStoredProcedureQuery Annotation

```
@NamedStoredProcedureQuery(name="findAllEmployees", procedureName="EMP_READ_ALL",
resultClass=Employee.class, parameters={
    @StoredProcedureParameter(queryParameter="result", name="RESULT_CURSOR",
direction=Direction.OUT_CURSOR})
@Entity
public class Employee {
 ...
}
```

Example 2–65 shows how to use the <named-stored-procedure-query> element in the eclipselink-orm.xml file.

Example 2–65 Using <named-stored-procedure-query> XML

```
<named-stored-procedure-query name="SProcXMLInOut" result-class="Address"
procedure-name="SProc_Read_XMLInOut">
    <parameter direction="IN_OUT" name="address_id_v" query-parameter="ADDRESS_ID"
type="Long"/>
    <parameter direction="OUT" name="street_v" query-parameter="STREET"
type="String"/>
</named-stored-procedure-query>
```

See Also

For more information, see:

- "@NamedStoredProcedureQueries" on page -94
- "Stored Procedures Examples" http://wiki.eclipse.org/EclipseLink/Examples/JPA/StoredProcedures

@Noncacheable

Use `@Noncacheable` to configure caching behavior for relationships. If used on a relationship, that relationship *will not* be cached, even though the parent Entity may be cached.

Annotation Elements

There are no elements for this annotation.

Usage

Each time EclipseLink retrieves the Entity, the relationship will be reloaded from the datasource. This may be useful for situations where caching of relationships is not desired or when using different EclipseLink IdentityMap types and having cached references extends the cache lifetime of related Entities using a different caching scheme. For instance Entity A references Entity B, Entity A is FullIdentityMap and Entity B is WeakIdentityMap. Without removing the caching of the relationsip the Entity B's cache effectively become a FullIdentityMap.

Examples

Example 2–66 shows how to use `@Noncacheable` to create a protected cache.

Example 2–66 Using @Noncacheable Annotation

```
@Entity
@Cache(
  isolation=CacheIsolationType.PROTECTED
)
public class Employee {
  @Id
  private long id;
  ...
  @OneToMany(mappedBy="manager")
  @Noncacheable
  private List<Employee> managedEmployees;
  ...
}
```

Example 2–67 shows using the `<noncacheable>` XML element in the `eclipselink-orm.xml` file.

Example 2–67 Using <noncacheable> XML

```
<?xml version="1.0"?>
<entity-mappings
    xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.eclipse.org/eclipselink/xsds/persistence/orm
http://www.eclipse.org/eclipselink/xsds/eclipselink_orm_2_4.xsd"
    version="2.4">
    <entity name="Employee" class="org.acme.Employee" access="FIELD">
        <cache isolation="PROTECTED"/>
        <attributes>
            <id name= "id"/>
```

```
            <one-to-many name="managedEmployees" mapped-by="manager">
                <noncacheable/>
            </one-to-many>
        </attributes>
    </entity>
</entity-mappings
```

See Also

For more information, see:

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

@NoSql

Use `@NoSql` to specify a non-relational (that is, no SQL) data source. EclipseLink can map non-relational data to objects and access that data through JPA.

Annotation Elements

Table 2–39 describes this annotation's elements.

Table 2–39 @NoSql Annotation Elements

Annotation Element	Description	Default
`dataFormat`	(Optional) Defines the order of the fields contained in the database structure type.	
`dataType`	(Required) The database name of the database structure type. ■ `INDEXED` – Uses JCA `IndexedRecords`. Object data is decomposed into an array of field values. ■ `MAPPED` – Uses JCA `MappedRecords`. Object data is decomposed into a map of key/value pairs. ■ `XML` – Uses XML data	XML

Usage

EclipseLink supports access to NoSQL data through the JavaEE Connector Architecture. You must use a JCA adapter (provided by EclipseLink, a third party, or custom built).

EclipseLink can use non-relational data sources such as:

- NoSQL databases such as Oracle NoSQL, MongoDB, and so on
- XML databases
- Distributed cache stores, such as Oracle Coherence
- Object databases
- Legacy databases, gateways and transaction system such as VSAM, ADA, CICS, IMS, MQSeries, Tuxedo, and so on
- ERP systems, such as SAP

You can map Entity and Embeddable objects to NoSQL data.

Most NoSQL data is hierarchical in form so using embeddable objects is common. Some NoSQL adaptors support XML data, so NoSQL mapped objects can use XML mappings when mapping to XML.

Examples

Example 2–68 shows using `@NoSql` with an XML data source.

Example 2–68 Using @NoSql Annotation

```
@Entity
@NoSQL(dataType="order")
public class Order {
  @Id
  @GeneratedValue
```

```
  @Field(name="@id")
  private long id;
  @Basic
  @Field(name="@description")
  private String description;
  @Embedded
  @Field(name="delivery-address")
  private Address deliveryAddress
  @ElementCollection
  @Field(name="orderLines/order-line")
  private List<OrderLine> orderLines;
  @ManyToOne
  @JoinField(name="customer-id")
  private Customer customer;
}

@Embeddable
@NoSQL(dataFormat=DataFormatType.MAPPED)
public class OrderLine {
    @Field(name="@line-number")
    private int lineNumber;
    @Field(name="@item-name")
    private String itemName;
    @Field(name="@quantity")
    private int quantity;
}
```

This would produce the following XML data:

```
<order id="4F99702B271B1948027FAF06" description="widget order">
  <deliveryAddress street="1712 Hasting Street" city="Ottawa" province="ON"
postalCode="L5J1H5"/>
  <order-lines>
      <order-line lineNumber="1" itemName="widget A" quantity="5"/>
      <order-line lineNumber="2" itemName="widget B" quantity="1"/>
      <order-line lineNumber="3" itemName="widget C" quantity="2"/>
  <order-lines>
  <customer-id>4F99702B271B1948027FAF08</customer-id>
<order>
```

See Also

For more information, see:

- @NoSQL
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/NoSQL/Configuring
- NoSQL Persistence Units
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/NoSQL/Persistence_Units
- Examples
  http://wiki.eclipse.org/EclipseLink/Examples/JPA/NoSQL

@ObjectTypeConverter

The `@ObjectTypeConverter` annotation specifies an `org.eclipse.persistence.mappings.converters.ObjectTypeConverter` that converts a fixed number of database data value(s) to Java object value(s) during the reading and writing of a mapped attribute.

Annotation Elements

Table 2–40 describes this annotation's elements.

Table 2–40 @ObjectTypeConverter Annotation Elements

Annotation Element	Description	Default
`name`	Set this attribute to the `String` name for your converter. Ensure that this name is unique across the persistence unit	none
`dataType`	(Optional) Set this attribute to the type stored in the database.	`void.class`[1]
`objectType`	(Optional) Set the value of this attribute to the type stored on the entity.	`void.class`[1]
`conversionValues`	Set the value of this attribute to the array of conversion values (instances of `ConversionValue: String objectValue` and `String dataValue`.	none
`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`

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

Usage

EclipseLink also includes `@TypeConverter` and `@StructConverter` converters.

Examples

Example 2–69 shows how to use the `@ObjectTypeConverter` annotation to specify object converters for the `gender` field.

Example 2–69 Using 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;
     }
     ...
 }
```

You can use the `<object-type-converter>` element in the deployment descriptor as an alternative to using the `@ObjectTypeConverter` annotation in the source code, as shown in Example 2–70.

Example 2–70 Using <object-type-converter> XML

```
<object-type-converter name="gender-converter" object-type="model.Gender"
    data-type="java.lang.String">
    <conversion-value object-value="Male" data-value="M" />
    <conversion-value object-value="Female" data-value="F" />
</object-type-converter>
```

See Also

For more information, see:

- "@TypeConverter" on page -170
- "@StructConverter" on page -150
- "@ConversionValue" on page -38

@ObjectTypeConverters

Use `@ObjectTypeConverters` to define multiple `ObjectTypeConverter` items.

Annotation Elements

Table 2–41 describes this annotation's elements.

Table 2–41 @ObjectTypeConverters Annotation Elements

Annotation Element	Description	Default
`ObjectTypeConverter`	(Required) An array of `@ObjectTypeConverter`	

Examples

Example 2–71 shows how to use this annotation.

Example 2–71 Using @ObjectTypeConverters Annotation

```
@Entity(name="Employee")
@Table(name="CMP3_FA_EMPLOYEE")
@ObjectTypeConverters({
  @ObjectTypeConverter(
    name="sex",
    dataType=String.class,

objectType=org.eclipse.persistence.testing.models.jpa.fieldaccess.advanced.Employe
e.Gender.class,
    conversionValues={
      @ConversionValue(dataValue="F", objectValue="Female"),
      @ConversionValue(dataValue="M", objectValue="Male")
    }
  )
})
```

To define multiple object type converts in the `eclipselink-orm.xml` file, simply create a list of multiple `<object-type-converter>` elements.

See Also

For more information, see:

- "@ObjectTypeConverter" on page -102

@OptimisticLocking

Use `@OptimisticLocking` to specify the type of optimistic locking EclipseLink should use when updating or deleting entities.

Annotation Elements

Table 2–42 describes this annotation's elements.

Table 2–42 @OptimisticLocking Annotation Elements

Annotation Element	Description	Default
`cascade`	(Optional) Specify where the optimistic locking policy should cascade lock. Currently only supported with `VERSION_COLUMN` locking.	`false`
`SelectedColumns`	(Optional) Specify a list of columns that will be optimistically locked. This element is required when `type=SELECTED_COLUMNS`.	
`type`	(Optional) The type of optimistic locking policy to use: ■ `ALL_COLUMNS` – EclipseLink compares every field in the table with the `WHERE` clause, when performing and update or delete operation. ■ `CHANGED_COLUMNS` – EclipseLink compares only the changed fields in the `WHERE` clause when performing an update. ■ `SELECTED_COLUMNS` – EclipseLink compares the selected field in the `WHERE` clause when performing and update or delete operation on the `SelectedColumns`. ■ `VERSION_COLUMN` – EclipseLink compares a single version number in the `WHERE` clause when performing an update.	`VERSION_COLUMN`

Usage

You can specify `@OptimisticLocking` on an Entity or MappedSuperclass.

Examples

Example 2–72 shows how to use the `@OptimisticLocking` annotation for all columns

Example 2–72 Using @OptimisticLocking Annotation

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

Example 2–72 shows how to use the `<optimistic-locking>` element in the `eclipselink-orm.xml` file for a single column.

Example 2–73 Using <optimistic-locking> XML

```
<entity name="Employee" class="my.Employee" access="PROPERTY"
change-tracking="DEFERRED">
...
    <optimistic-locking type="SELECTED_COLUMNS" cascade="false">
```

```
      <selected-column name="id"/>
      <selected-column name="firstName"/>
    </optimistic-locking>
...
</entity>
```

See Also

For more information, see:

@OrderCorrection

Use `@OrderCorrection` to specify a strategy to use if the order list read from the database is invalid (for example, it has nulls, duplicates, negative values, or values greater than or equal to the list site).

To be valid, an order list of *n* elements must be `{0, 1,..., n-1}`

Annotation Elements

Table 2–43 describes this annotation's elements.

Table 2–43 @OrderCorrection Annotation Elements

Annotation Element	Description	Default
`OrderCorrectionType`	(Optional) Specify a strategy to use if the order list read from the database is invalid: ■ `EXCEPTION` ■ `READ` ■ `READ_WRITE`	`READ_WRITE`

Usage

When using `@OrderCorrection`, you can specify how EclipseLink should handle invalid list orders:

- `EXCEPTION` – When `OrderCorrectionType=EXCEPTION`, EclipseLink will not correct the list. Instead, EclipseLink will throw a `QueryException` with error code `QueryException.LIST_ORDER_FIELD_WRONG_VALUE`

  For example, given the following list of three objects in the database:

  ```
  {null, objectA}; {2, objectB}, {5, ObjectC};
  ```

  When read into the application, EclipseLink will throw an exception.

- `READ` – When `OrderCorrectionType=READ`, EclipseLink corrects the list read into application, but does not retain any information about the invalid list order that remains in the database. Although this is not an issue in read-only uses of the list, if the list is modified and then saved into the database, the order will most likely differ from the cache and be invalid.

  The `READ` mode is used as the default when the mapped attribute is neither a `List` nor `Vector` (that is, it is not assignable from the EclipseLink internal class `IndirectList`).

  For example, given the following list of three objects in the database:

  ```
  {null, objectA}; {2, objectB}, {5, ObjectC}
  ```

  - When read as a list: `{objectA, objectB, objectC}`
  - When adding a new element to the list: `{objectA, objectB, objectC, objectD}`
  - When saving the updated list to the database: `{null, objectA}, {2, objectB}, {5, objectC}, {3, objectD}`
  - When reading the list again: `{objectA, objectB, objectD, objectC}`

- `READ_WRITE` – When `OrderCorrectionType=READ_WRITE`, EclipseLink corrects the order of the list read into application *and* remembers the invalid list order left in the database. If the list is updated and saved to the database, the order indexes are saved ensuring that the list order in the data base will be exactly the same as in cache (and therefore valid).

  The `READ_WRITE` mode is used as the default when the mapped attribute is either a `List` or `Vector` (that is, it is assignable from the EclipseLink internal class `IndirectList`). In JPA, if the mode is not specified, `READ_WRITE` is used by default.

  For example, given the following list of three objects in the database:

  ```
  {null, objectA}; {2, objectB}, {5, ObjectC}
  ```

  - When read as a list: `{objectA, objectB, objectC}`
  - When adding a new element to the list: `{objectA, objectB, objectC, objectD}`
  - When saving the updated list to the database: `{0, objectA}, {1, objectB}, {2, objectC}, {3, objectD}`
  - When reading the list again: `{objectA, objectB, objectC, objectD}`

Examples

Example 2–74 shows how to use this annotation.

Example 2–74 Using @OrderCorrection Annotation

```
@OrderColumn(name="ORDER_COLUMN")
@OrderCorrection(EXCEPTION)
List<String> designations;
```

See Also

For more information see:

- "Entity Annotations" on page 2-2

@Partitioned

Use `@Partitioned` to specify a partitioning policy to use for an Entity or relationship.

Annotation Elements

Table 2–44 describes this annotation's elements.

Table 2–44 @Partitioned Annotation Elements

Annotation Element	Description	Default
`value`	(Required) Name of the partitioning policy: ■ @HashPartitioning ■ @PinnedPartitioning ■ @RangePartition ■ @ReplicationPartitioning ■ @RoundRobinPartitioning ■ @UnionPartitioning ■ @ValuePartitioning	

Usage

Use partitioning to partition the data for a class across multiple databases or a database cluster (such as Oracle RAC). Partitioning can provide improved scalability by allowing multiple database machines to service requests.

You can specify `@Partitioned` on an Entity, relationship, query, or session/persistence unit.

Examples

Example 2–75 shows the partition of `Employee` data by location. The two primary sites, Ottawa and Toronto, are stored on separate databases. All other locations are stored on the default database.

The example project is range partitioned by its ID. Each range of ID values are stored on a different database. The employee/project relationship is an example of a cross partition relationship.

To allow the employees and projects to be stored on different databases, EclipseLink uses a union policy; the join table is replicated to each database.

Example 2–75 Using Partitioning

```
@Entity
@IdClass(EmployeePK.class)
@UnionPartitioning(
        name="UnionPartitioningAllNodes",
        replicateWrites=true)
@ValuePartitioning(
        name="ValuePartitioningByLOCATION",
        partitionColumn=@Column(name="LOCATION"),
        unionUnpartitionableQueries=true,
        defaultConnectionPool="default",
        partitions={
            @ValuePartition(connectionPool="node2", value="Ottawa"),
```

```
            @ValuePartition(connectionPool="node3", value="Toronto")
        })
@Partitioned("ValuePartitioningByLOCATION")
public class Employee {
    @Id
    @Column(name = "EMP_ID")
    private Integer id;

    @Id
    private String location;
    ...

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

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

See Also

For more information, see:

- "@Partitioning"
- "@HashPartitioning" on page -66
- "@PinnedPartitioning" on page -118
- "@RangePartition" on page -130
- "@ReplicationPartitioning" on page -138
- "@RoundRobinPartitioning" on page -144
- "@UnionPartitioning" on page -178
- "@ValuePartitioning" on page -182

- "Data Partitioning"
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning
- Partitioning Examples
  http://wiki.eclipse.org/EclipseLink/Examples/JPA/Partitioning

@Partitioning

Use `@Partitioning` to configure a custom `PartitioningPolicy`.

Annotation Elements

Table 2–45 describes this annotation's elements.

Table 2–45 @Partitioning Annotation Elements

Annotation Element	Description	Default
`name`	Name of the partition policy. Names must be unique for the persistence unit.	
`PartitioningClass`	(Required) Full `package.class` name of a subclass of `PartitioningPolicy`.	

Usage

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

Partitioning can be enabled on an entity, a relationship, a query, or a persistence unit.

Partitioning Policies

To configure data partitioning, use the `@Partitioned` annotation and one or more partitioning policy annotations. The annotations for defining the different kinds of policies are:

- `@HashPartitioning`: Partitions access to a database cluster by the hash of a field value from the object, such as the object's ID, location, or tenant. The hash indexes into the list of connection pools/nodes. All write or read request for objects with that hash value are sent to the same server. If a query does not include the hash field as a parameter, it can be sent to all servers and unioned, or it can be left to the session's default behavior.
- `@PinnedPartitioning`: Pins requests to a single connection pool/node. This allows for vertical partitioning.
- `@RangePartitioning`: Partitions access to a database cluster by a field value from the object, such as the object's ID, location, or tenant. Each server is assigned a range of values. All write or read requests for objects with that value are sent to the same server. If a query does not include the field as a parameter, then it can either be sent to all server's and unioned, or left to the session's default behavior.
- `@ReplicationPartitioning`: Sends requests to a set of connection pools/nodes. This policy is for replicating data across a cluster of database machines. Only modification queries are replicated.
- `@RoundRobinPartitioning`: Sends requests in a round-robin fashion to the set of connection pools/nodes. It is for load balancing read queries across a cluster of database machines. It requires that the full database be replicated on each machine, so it does not support partitioning. The data should either be read-only, or writes should be replicated.

- `@UnionPartitioning`: Sends queries to all connection pools and unions the results. This is for queries or relationships that span partitions when partitioning is used, such as on a `ManyToMany` cross partition relationship.
- `@ValuePartitioning`: Partitions access to a database cluster by a field value from the object, such as the object's location or tenant. Each value is assigned a specific server. All write or read requests for objects with that value are sent to the same server. If a query does not include the field as a parameter, then it can be sent to all servers and unioned, or it can be left to the session's default behavior.
- `@Partitioning`: Partitions access to a database cluster by a custom partitioning policy. A PartitioningPolicy class must be provided and implemented.

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

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

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

Clustered Databases and Oracle RAC

Some databases support clustering the database across multiple machines. Oracle RAC allows for a single database to span multiple different server nodes. Oracle RAC also supports table and node partitioning of data. A database cluster allows for any of the data to be accessed from any node in the cluster. However, it is generally it is more efficient to partition the data access to specific nodes, to reduce cross node communication.

EclipseLink partitioning can be used in conjunction with a clustered database to reduce cross node communication, and improve scalability.

To use partitioning with a database cluster to following is required:

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

Examples

Example 2–76 shows how to partition Employee data by location. The two primary sites, **Ottawa** and **Toronto** are each stored on a separate database. All other locations are stored on the default database. Project is range partitioned by its ID, as shown in Example 2–77. Each range of ID values are stored on a different database. The employee/project relationship is an example of a cross partition relationship. To allow

the employees and projects to be stored on different databases a union policy is used and the join table is replicated to each database.

Example 2–76 Using Partitioning

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

    @Id
    private String location;
    ...

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

Example 2–77 Using @RangePartitioning

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

See Also

For more information, see:

- "@Partitioned" on page -110
- "@HashPartitioning" on page -66
- "@PinnedPartitioning" on page -118
- "@RangePartitioning" on page -132
- "@ReplicationPartitioning" on page -138
- "@RoundRobinPartitioning" on page -144
- "@UnionPartitioning" on page -178
- "@ValuePartitioning" on page -182
- "Data Partitioning" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning
- EclipseLink Examples http://wiki.eclipse.org/EclipseLink/Examples/JPA/Partitioning

@PinnedPartitioning

Use `@PinnedPartitionPolicy` to pin requests to a single connection pool.

Annotation Elements

Table 2–46 describes this annotation's elements.

Table 2–46 @PinnedPartitioning Annotation Elements

Annotation Element	Description	Default
`connectionPool`	Connection pool name to which to pin queries.	
`name`	Name of the partition policy. Names must be unique for the persistence unit.	

Usage

Partition policies are globally named, to allow reuse. You must also set the partitioning policy with the `@Partitioned` annotation.

You can specify `@PinnedPartitioning` on an Entity, relationship, query, or session/persistence unit.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

If a transaction modifies data from multiple partitions, you should use JTA ensure proper two-phase commit of the data. You can also configure an exclusive connection in the EntityManager to ensure that only a single node is used for a single transaction.

Examples

See Example 2–75 on page -110 for an example of partitioning with EclipseLink.

See Also

For more information, see:

- "Data Partitioning" `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning`
- "@Partitioned" on page -110

@PrimaryKey

Use `@PrimaryKey` to allow advanced configuration of the ID.

A validation policy can be given that allows specifying if zero is a valid ID value. The set of primary key columns can also be specified precisely.

Annotation Elements

Table 2–47 describes this annotation's elements.

Table 2–47 @PrimaryKey Annotation Elements

Annotation Element	Description	Default
`cacheKeyType`	(Optional) Configures the cache key type to store the object in the cache.	`AUTO`
`columns`	(Optional) Directly specify the primary key columns. This can be used instead of `@Id` if the primary key includes a non basic field, such as a foreign key, or a inheritance discriminator, embedded, or transformation mapped field.	
`validation`	(Optional) Configures what ID validation is done: ■ `NULL` – EclipseLink interprets zero values as zero. This permits primary keys to use a value of zero. ■ `ZERO` (default) – EclipseLink interprets zero as null. ■ `NEGATIVE` – EclipseLink interprets negative values as null. ■ `NONE` – EclipseLink does not validate the id value. By default `0` is not a valid ID value, this can be used to allow 0 id values.	`ZERO`

Usage

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 `@PrimaryKey` annotation to configure an `IdValidation` for an entity class.

Examples

Example 2–78 shows how to use this annotation.

Example 2–78 Using @PrimaryKey Annotation

```
@PrimaryKey(validation=IdValidation.ZERO)
public class Employee implements Serializable, Cloneable {
...
}
```

Example 2–79 shows how to use the `<primary-key>` element in your `eclipselink-orm.xml` file.

Example 2–79 Using @<primary-key> XML

```
<entity name="Employee" class="foo.Employee" access="PROPERTY">
   <primary-key validation="ZERO"/>
...
</entity>
```

See Also

For more information, see:

- "@Id" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/Ids/Id
- "Entity Annotations" on page 2-2

@PrivateOwned

Use `@PrivateOwned` to specify that a relationship is privately owned; target object is a dependent part of the source object and is not referenced by any other object and cannot exist on its own.

Annotation Elements

The `@PrivateOwned` annotation does not have attributes.

Usage

Using `@PrivateOwned` causes many operations to be cascaded across the relationship including delete, insert, refresh, and lock (when cascaded). It also ensures that private objects removed from collections are deleted and that objects added are inserted.

You can specify `@PrivateOwned` on with `@OneToOne`, `@OneToMany` and `@VariableOneToOne` annotations. Private ownership is implied with the `@BasicCollection` and `@BasicMap` annotations.

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

Additional Information

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

- If the source of a privately owned relationship is deleted, then EclipseLink will delete the target. This is equivalent of setting @CascadeOnDelete.
- If you remove the reference to a target from a source, then EclipseLink will delete the target.

Normally, do not configure privately owned relationships on 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.

Examples

Example 2–80 shows using `@PrivateOwned` to specify `Employee` field `phoneNumbers`. .

Example 2–80 Using @PrivateOwned Annotation

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

When you create a mapping using Java, use its `privateOwnedRelationship` method to tell EclipseLink that the referenced object is privately owned: that is, the referenced child object cannot exist without the parent object.

See Also

For more information, see:

- "@CascadeOnDelete" on page -26

@Properties

Use `@Property` to specify a single user-defined property on a mapped attribute or its `get/set` method. Use the `@Properties` annotation to wrap multiple properties.

Annotation Elements

Table 2–48 describes this annotation's elements.

Table 2–48 @Properties Annotation Elements

Annotation Element	Description	Default
Property	Array of `Property` elements.	

Usage

You can specify `@Property` on a mapped attribute (or its `get/set` method) within an Entity, MappedSuperclass, or Embeddable class. You can also specify this annotation on an Entity, MappedSuperclass, or Embeddable class.

Properties defined in MappedSuperclass are passed to all inheriting Entities and MappedSuperclasses. In case of a conflict, property values defined directly on a class always override values inherited from a class's parent.

When using an `orm.xml` mapping file, EclipseLink ignores `@Property` and `@Properties` annotations on mapped attributes; annotations on classes are merged with those specified i the `orm.xml` file, with the latter taking precedence in case of conflicts.

Examples

Example 2–101 on page -168 shows how to use the `@Properties` annotation within a `@Transformation` mapping. Example 2–102 shows how to use the `<properties>` XML element within the `orm.xml` file.

See Also

For more information, see:

- "@Property" on page -126

@Property

Use `@Property` to specify a single user-defined property on a mapped attribute or its `get/set` method. Use the `@Properties` annotation to wrap multiple properties.

Annotation Elements

Table 2–49 describes this annotation's elements.

Table 2–49 @Property Annotation Elements

Annotation Element	Description	Default
`name`	(Required) Name of the property	
`value`	(Required) String representation of the property `value`, converted to an instance of `valueType`.	
`valueType`	(Optional) Property value type, converted to valueType by `ConversionManager`. This must be a simple type that can be handled by the `ConversionManager`.	`String`

Usage

You can specify `@Property` on a mapped attribute (or its `get/set` method) within an Entity, MappedSuperclass, or Embeddable class. You can also specify this annotation on an Entity, MappedSuperclass, or Embeddable class.

Properties defined in MappedSuperclass are passed to all inheriting Entities and MappedSuperclasses. In case of a conflict, property values defined directly on a class always override values inherited from a class's parent.

When using an `orm.xml` mapping file, EclipseLink ignores `@Property` and `@Properties` annotations on mapped attributes; annotations on classes are merged with those specified i the `orm.xml` file, with the latter taking precedence in case of conflicts.

Examples

Example 2–101 on page -168 shows how to use the `@Property` annotation within a `@Transformation` mapping. Example 2–102 shows how to use the `<property>` XML element within the `orm.xml` file.

See Also

For more information, see:

- "@Properties" on page -124

@QueryRedirectors

Use `@QueryRedirectors` to intercept EclipseLink queries for pre- and post-processing, redirection, or performing some side effect such as auditing.

Annotation Elements

Table 2–50 describes this annotation's elements.

Table 2–50 @QueryRedirectors Annotation Elements

Annotation Element	Description	Default
`allQueries`	This `AllQueries` Query Redirector will be applied to any executing object query that does not have a more precise redirector (like the `ReadObjectQuery` Redirector) or a redirector set directly on the query.	`void.class`
`delete`	A Default `Delete` Object Query Redirector will be applied to any executing `DeleteObjectQuery` or `DeleteAllQuery` that does not have a redirector set directly on the query.	`void.class`
insert	A Default `Insert` Query Redirector will be applied to any executing `InsertObjectQuery` that does not have a redirector set directly on the query.	`void.class`
readAll	A Default `ReadAll` Query Redirector will be applied to any executing `ReadAllQuery` that does not have a redirector set directly on the query. For users executing a JPA Query through the `getResultList()`, API this is the redirector that will be invoked	`void.class`
`readObject`	A Default `ReadObject` Query Redirector will be applied to any executing `ReadObjectQuery` that does not have a redirector set directly on the query. For users executing a JPA Query through the `getSingleResult()` API or `EntityManager.find()`, this is the redirector that will be invoked	`void.class`
`report`	A Default `ReportQuery` Redirector will be applied to any executing `ReportQuery` that does not have a redirector set directly on the query. For users executing a JPA Query that contains aggregate functions or selects multiple entities this is the redirector that will be invoked	`void.class`
`update`	A Default `Update` Query Redirector will be applied to any executing UpdateObjectQuery or `UpdateAllQuery` that does not have a redirector set directly on the query. In EclipseLink an `UpdateObjectQuery` is executed whenever flushing changes to the datasource.	`void.class`

Usage

Use `@QueryRedirectors` to extend the standard EclipseLink query functionality.

You can set a `QueryRedirector` through the Query Hint `eclipselink.query.redirector` or set as a default Redirector on an Entity.

Examples

Example 2–81 shows how to use this annotation.

Example 2–81 Using @QueryRedirectors Annotation

```
@QueryRedirectors(
    allQueries=org.queryredirectors.AllQueriesForEntity.class)
@Entity
public class
...
```

See Also

For more information, see:

@RangePartition

Use `@RangePartition` to create a specific range partition for a connection pool. Values within the range will be routed to the specified connection pool.

Annotation Elements

Table 2–51 describes this annotation's elements.

Table 2–51 @RangePartition Annotation Elements

Annotation Element	Description	Default
`connectionPool`	The connection pool to which to route queries for the specified range.	
`startValue`	The `String` representation of the range start value.	
`endValue`	The `String` representation of the range end value.	

Examples

See Example 2–75 on page -110 for an example of partitioning with EclipseLink.

See Also

For more information, see:

- "Data Partitioning" `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning`
- "@Partitioned" on page -110

@RangePartitioning

Use `@RangePartitioning` to partitions access to a database cluster by a field value from the object (such as the object's ID, location, or tenant).

EclipseLink assigns each server a range of values. All write or read request for objects with a server's value are sent to that specific server. If a query does not include the field as a parameter, then it can either be sent to all server's and unioned, or left to the session's default behavior.

Annotation Elements

Table 2–52 describes this annotation's elements.

Table 2–52 @RangePartitioning Annotation Elements

Annotation Element	Description	Default
`name`	(Required) The name of the partition policy; must be unique for the persistence unit.	
`partitionColumn`	(Required) The database column or query parameter to partition queries by. This is the *table column name*, not the class attribute name. The column value must be included in the query and should normally be part of the object's ID. This can also be the name of a query parameter. If a query does not contain the field the query will not be partitioned.	
`partitions`	(Required) List of connection pool names to load balance across.	
`partitionValueType`	The type of the start and end values.	`String`
`unionunpartionableQueries`	Defines if queries that do not contain the partition field should be sent to every database and have the result unioned.	`false`

Usage

Partitioning can be enabled on an Entity, relationship, query, or session/persistence unit.

Partition policies are globally named to allow reuse, the partitioning policy must also be set using the `@Partitioned` annotation to be used.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

If a transaction modifies data from multiple partitions, you should use JTA ensure proper two-phase commit of the data. You can also configure an exclusive connection in the EntityManager to ensure that only a single node is used for a single transaction.

Examples

Example 2–82 shows how to use the `@RangePartitioning` annotation

Example 2–82 Using @RangePartitioning Annotation

```
@Entity
@Table(name="PART_PROJECT")
@RangePartitioning(
```

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

Example 2–82 shows how to use the `<range-partitioning>` element in the `eclipselink-orm.xml` file.

Example 2–83 Using <range-partitioning> XML

```
<entity name="Project" class="Project" access="FIELD">
  <table name="PART_PROJECT"/>
  <range-partitioning name="RangePartitioningByPROJ_ID"
partition-value-type="java.lang.Integer" union-unpartitionable-queries="true">
    <partition-column name="PROJ_ID"/>
    <partition connection-pool="default" start-value="0" end-value="1000"/>
    <partition connection-pool="node2" start-value="1000" end-value="2000"/>
    <partition connection-pool="node3" start-value="2000"/>
  </range-partitioning>
  <partitioned>RangePartitioningByPROJ_ID</partitioned>
</entity>
```

See Also

For more information, see:

- "@RangePartition" on page -130
- "@Partitioned" on page -110

@ReadOnly

Use `@ReadOnly` to specify that a class is read-only.

Annotation Elements

This annotation contains no elements.

Usage

It may be defined on an Entity or MappedSuperclass.

In the case of inheritance, a `@ReadOnly` annotation can only be defined on the root of the inheritance hierarchy .

You can also use `@ReadOnly` to bypass EclipseLink's persistence context to save heap space (such as if you need to load a large dataset).

Examples

Example 2–84 shows how to use this annotation.

Example 2–84 Using @ReadOnly Annotation

```
@ReadOnly
@Entity
@Table(name = "TMP_READONLY")
public class ReadOnlyEntity {
...
}
```

Example 2–85 shows how to use the `<read-only>` element in the `eclipselink-orm.xml` file.

Example 2–85 Using <read-only> XML

```
<entity name="XMLReadOnlyClass" class="ReadOnlyClass" access="PROPERTY"
read-only="true">
```

See Also

For more information, see:

- "Entity Annotations" on page 2-2

@ReadTransformer

Use `@ReadTransformer` with Transformation mappings to define the transformation of the database column values into attribute values (unless the mapping is write-only).

Annotation Elements

Table 2–53 describes this annotation's elements.

Table 2–53 @ReadTransformer Annotation Elements

Annotation Element	Description	Default
`method`	The mapped class must have a method with this name which returns a value to be assigned to the attribute (not assigns the value to the attribute).	
`transformerClass`	User-defined class that implements the `org.eclipse.persistence.mappings.transformers.AttributeTransformer` interface. The class will be instantiated, its `buildAttributeValue` will be used to create the value to be assigned to the attribute.	`void.class`

Note: You must specify **either** a `method` or `transformerClass`, but not both.

Usage

Also unless it's a read-only mapping, either `@WriteTransformer` annotation or `@WriteTransformers` annotation should be specified. Each `WriteTransformer` defines transformation of the attribute value to a single database column value (column is specified in the `WriteTransformer`).

Examples

See "Using @Transformation Annotation" on page -168 for an example of how to use the `@WriteTransformer annotation` with a Transformation mapping.

See Also

For more information, see:

- "How to Use the @ReadTransformer Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40ReadTransformer_Annotation
- "@Transformation" on page -168.

@ReplicationPartitioning

Use `@ReplicationPartitioning` to send requests to a set of connection pools. It is for replicating data across a cluster of database machines. Only modification queries are replicated.

Annotation Elements

Table 2–54 describes this annotation's elements.

Table 2–54 @ReplicationPartitioning Annotation Elements

Annotation Element	Description	Default
`name`	The name of the partition policy; must be unique for the persistence unit.	
`connectionPools`	List of connection pool names to load balance across.	All defined pools in the `ServerSession`

Usage

Partitioning can be enabled on an Entity, relationship, query, or session/persistence unit.

Partition policies are globally named to allow reuse, the partitioning policy must also be set using the `@Partitioned` annotation to be used.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

If a transaction modifies data from multiple partitions, you should use JTA ensure proper two-phase commit of the data. You can also configure an exclusive connection in the EntityManager to ensure that only a single node is used for a single transaction.

Examples

See Example 2–75 on page -110 for an example of partitioning with EclipseLink.

See Also

For more information, see:

- "Data Partitioning" `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning`
- "@Partitioned" on page -110

@ReturnInsert

Use `@ReturnInsert` to cause `INSERT` operations to return values back into the object being written. This allows for table default values, trigger or stored procedures computed values to be set back into the object.

Annotation Elements

Table 2–55 describes this annotation's elements.

Table 2–55 @ReturnInsert Annotation Elements

Annotation Element	Description	Default
`returnOnly`	(Optional) If specified (true), the mapping field will be excluded from the `INSERT` clause during SQL generation.	false

Usage

A `@ReturnInsert` annotation can only be specified on a `Basic` mapping.

Examples

Example 2–86 shows how to use the `@ReturnInsert` annotation. If you do not use an argument, EclipseLink accepts the default value, `false`.

Example 2–86 Using @ReturnInsert Annotation

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

Example 2–87 shows how to use the `<return-insert>` element in the `eclipselink-orm.xml` file.

Example 2–87 Using <return-insert> XML

```
<basic name="firstName">
    <column name="FIRST_NAME"/>
    <return-insert read-only="true"/>
</basic>
```

See Also

For more information, see:

- "How to Use the @ReturnInsert Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40ReturnInsert_Annotation

@ReturnUpdate

Use `@ReturnUpdate` to cause `UPDATE` operations to return values back into the object being written. This allows for table default values, trigger or stored procedures computed values to be set back into the object.

Annotation Elements

This annotation contains no elements.

Usage

A `@ReturnUpdate` annotation can only be specified on a `Basic` mapping.

Examples

Example 2–88 shows how to use the `@ReturnUpdate` annotation. The annotation does not accept any arguments.

Example 2–88 Using @ReturnUpdate Annotation

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

Example 2–89 illustrates the same example as before, but uses the `<return-update>` element in the `eclipselink-orm.xml` mapping file.

Example 2–89 Using <return-update> XML

```
<basic name="firstName">
    <column name="F_NAME"/>
    <return-update/>
</basic>
```

See Also

For more information, see:

- "How to Use the @ReturnInsert Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40ReturnInsert_Annotation
- "@ReturnInsert" on page -140

@RoundRobinPartitioning

Use `@RoundRobinPartitioning` to send requests in a "round robin" fashion to the set of connection pools.

Annotation Elements

Table 2–56 describes this annotation's elements.

Table 2–56 @RoundRobinPartitioning Annotation Elements

Annotation Element	Description	Default
`name`	(Required) Name of the partition policy. Names must be unique for the persistence unit.	
`connectionPools`	(Optional) List of connection pool names to load balance across.	All defined pools in the `ServerSession`
`replicateWrite`	(Optional) This allows for a set of database to be written to and kept in sync, and have reads load-balanced across the databases.	`false`

Usage

Use the `@RoundRobinPartitioning` annotation for load-balancing read queries across a cluster of database machines. Using `@RoundRobinPartitioning` requires that the full database be replicated on each machine.

The data should either be read-only, or writes should be replicated on the database.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

If a transaction modifies data from multiple partitions, you should use JTA ensure proper two-phase commit of the data. You can also configure an exclusive connection in the EntityManager to ensure that only a single node is used for a single transaction.

Examples

See "@Partitioned" on page -110 for an example of partitioning with EclipseLink.

See Also

For more information, see:

- "Data Partitioning" `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning`
- "@Partitioned" on page -110

@StoredProcedureParameter

Use `@StoredProcedureParameter` within a `NamedStoredProcedureQuery` annotation..

Annotation Elements

Table 2–57 describes this annotation's elements.

Table 2–57 @StoredProcedureParameter Annotation Elements

Annotation Element	Description	Default
`queryParameter`	(Required) The query parameter name	
`direction`	(Optional) The direction of the stored procedure parameter: ■ `IN` – Input parameter ■ `IN_OUT` – Input and output parameters ■ `OUT` – Output parameter ■ `OUT_CURSOR` – Output cursor	`IN`
`jdbcType`	(Optional) JDBC type code. This depends on the type returned from the procedure.	`-1`
`jdbcTypeName`	(Optional) JDBC type name. This may be required for `ARRAY` or `STRUCT` types.	
`name`	(Optional) Stored procedure parameter name	
`optional`	(Optional) Specify if the parameter is required, or optional and defaulted by the procedure.	`false`
`type`	(Optional) Type of Java class desired back from the procedure. This depends on the `type` returned from the procedure.	`void.class`

Usage

EclipseLink will throw an exception if you set more than one parameter to the `OUT_CURSOR` type.

Examples

See "@NamedStoredProcedureQuery" on page -96 for an example using the `@StoredProcedureParameter` annotation.

See Also

For more information:

- "@NamedStoredProcedureQuery" on page -96
- Stored Procedure Examples
  http://wiki.eclipse.org/EclipseLink/Examples/JPA/StoredProcedures

@Struct

Use @Struct to define a class to map to a database `Struct` type. The class should normally be an Embeddable, but could also be an Entity if stored in a object table.

Annotation Elements

Table 2–58 describes this annotation's elements.

Table 2–58 @Struct Annotation Elements

Annotation Element	Description	Default
`name`	(Required) The database name of the database structure type.	
`fields`	(Optional) Defines the order of the fields contained in the database structure type.	

Usage

`Struct` types are extended object-relational data-types supported by some databases. Struct types are user define types in the database such as `OBJECT` types on Oracle. Structs can normally contains Arrays (`VARRAY`) or other Struct types, and can be stored in a column or a table.

Examples

Example 2–90 shows using the `@Struct` annotation to define a Java class to map to an `OBJECT` type.

Example 2–90 Using @Struct Annotation

```
@Embeddable
@Struct(name="EMP_TYPE", fields={"F_NAME", "L_NAME", "SALARY"})
public class Employee {
 @Column(name="F_NAME")
 private String firstName;
 @Column(name="L_NAME")
 private String lastName;
 @Column(name="SALARY")
 private BigDecimal salary;
 ...
}
```

Example 2–91 shows how to use the `<struct>` element in the `eclipselink-orm.xml` file.

Example 2–91 Using <struct> XML

```
<embeddable class="Address" access="FIELD">
  <struct name="PLSQL_P_PLSQL_ADDRESS_REC">
    <field>ADDRESS_ID</field>
    <field>STREET_NUM</field>
    <field>STREET</field>
    <field>CITY</field>
    <field>STATE</field>
  </struct>
  <attributes>
```

```
    <basic name="id">
      <column name="ADDRESS_ID"/>
    </basic>
    <basic name="number">
      <column name="STREET_NUM"/>
    </basic>
  </attributes>
</embeddable>
```

See Also

For more information, see:

- "@Structure" on page -154

@StructConverter

The `@StructConverter` annotation is added to an `org.eclipse.persistence.platform.database.DatabasePlatform` using its `addStructConverter` method to enable custom processing of `java.sql.Struct` types.

Annotation Elements

Table 2–59 describes this annotation's elements.

Table 2–59 @StructConverter Annotation Elements

Annotation Element	Description	Default
`name`	The `String` name for your converter. Ensure that this name is unique across the persistence unit.	none
`converter`	The converter class as a `String`. This class must implement the `org.eclipse.persistence.mappings.converters.Converter` interface.	none

Usage

A `DatabasePlatform` object 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 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, the appropriate settings are applied to the mapping. This setting is 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.

EclipseLink also includes `@ObjectTypeConverter` and `@TypeConverter` converters.

Examples

Example 2–92 shows how to define the `@StructConverter` annotation.

Example 2–92 Using @StructConverter Annotation

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

You can specify the `@StructConverter` annotation anywhere in an Entity with the scope being the whole session. An exception is thrown if you add more than one `StructConverter` annotation that affects the same Java type. An `@StructConverter` annotation exists in the same namespaces as `@Converter`. A validation exception is

thrown if you add an `@Converter` and an `@StructConverter` of the same name. You can also configure structure converters in a `sessions.xml` file.

See Also

For more information, see:

- "@StructConverters" on page -152
- "Default Conversions and Converters" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Default_Conversions_and_Converters/StructConverter

@StructConverters

Use `@StructConverters` to define multiple `@StructConverter` annotations.

Annotation Elements

Table 2–60 describes this annotation's elements.

Table 2–60 @StructConverters Annotation Elements

Annotation Element	Description	Default
`StructConverter[]`	(Required) An array of struct converter	

Examples

Example 2–93 shows how to use the `@StructConverters` annotation to define multiple `@StructConverter` elements.

Example 2–93 Using @StructConverters Annotation

```
@StructConverters{{
  @StructConverter(name="StructConverter1", converter="foo.StructConverter1"),
  @StructConverter(name="StructConverter2", converter="foo.StructConverter2")
  })
```

Example 2–94 shows how to use the `<struct-converters>` element in the `eclipselink-orm.xml` file.

Example 2–94 Using <struct-converters> XML

```
<struct-converters>
  <struct-converter name="StructConverter1" converter="foo.StructConverter1"/>
  <struct-converter name="StructConverter2" converter="foo.StructConverter2"/>
</struct-converters>
```

See Also

For more information, see:

- "@StructConverter" on page -150

@Structure

Use `@Structure` on a field/method to define an `StructureMapping` to an embedded `Struct` type. The target Embeddable must be mapped using the Struct annotation.

Annotation Elements

This annotation contains no elements.

Usage

Struct types are extended object-relational data-types supported by some databases. Struct types are user define types in the database such as `OBJECT` types on Oracle. Structs can normally contains Arrays (`VARRAY`) or other Struct types, and can be stored in a column or a table.

Examples

Example 2–95 shows how to use the `@Structure` annotation. See Example 2–90 on page -154 to an example of using `@Struct` to map the target.

Example 2–95 Using @Structure Annotation

```
@Structure
protected Address address;
```

You can also define structure mappings in the `eclipselink-orm.xml` file by using the `<structure>` element.

Example 2–96 Using <structure> XML

```
<structure name="address"/>
```

See Also

For more information, see:

- "@Struct" on page -148

@TenantDiscriminatorColumn

The `@TenantDiscriminator` annotation is used with the `@Multitenant` annotation and the `SINGLE-TABLE` mulitenant type to limit what a persistence context can access in single-table mulitenancy.

Annotation Elements

Table 2–61 describes this annotation's elements.

Table 2–61 @TenantDiscriminatorColumn Properties

Annotation Element	Description	Default
`java.lang.String columnDefinition`	(Optional) The SQL fragment that is used when generating the DDL for the discriminator column.	The provider-generated SQL to create a column of the specified discriminator type.
`java.lang.String contextProperty`	(Optional) The name of the context property to apply to the tenant discriminator column.	`eclipselink.tenant-id`
`DiscriminatorType discriminatorType`	(Optional) The type of object/column to use as a class discriminator.	`javax.persistence.DiscriminatorType.STRING`
`int length`	(Optional) The column length for String-based discriminator types.	The column length for String-based discriminator types. Ignored for other discriminator types.
`java.lang.String name`	(Optional) The name of column to be used for the tenant discriminator.	`TENANT_ID`
`boolean primaryKey`	Specifies that the tenant discriminator column is part of the primary key of the tables.	`false`
`java.lang.String table`	(Optional) The name of the table that contains the column.	The name of the table that contains the column. If absent the column is assumed to be in the primary table. This attribute must be specified if the column is on a secondary table.

Usage

To configure single-table multi-tenancy, you must specify both of the following:

- Annotate the entity or mapped superclass to use single-table multi-tenancy, using the `@Multitenant` annotation, for example:

```
@Entity
@Table(name="EMP")
@Multitenant(SINGLE_TABLE)
```

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

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

- Specify the column or columns to be used as the discriminator column, using the `@TenantDiscriminatorColumn` annotation, for example:

```
@Entity
@Table(name="EMP")
@Multitenant(SINGLE_TABLE)
@TenantDiscriminatorColumn(name = "TENANT_ID")
```

  You can specify multiple discriminator columns by using the `@TenantDiscriminatorColumns` annotation, for example:

```
@Entity
@Table(name = "EMPLOYEE")
@Multitenant(SINGLE_TABLE)
@TenantDiscriminatorColumns({
    @TenantDiscriminatorColumn(name = "TENANT_ID")
    @TenantDiscriminatorColumn(name = "TENANT_CODE")})
```

Using Discriminator Columns

The following characteristics apply to discriminator columns:

- On persist, the values of tenant discriminator columns are populated from their associated context properties.
- Tenant discriminator columns are application definable. That is, the discriminator column is not tied to a specific column for each shared entity table. You can use TENANT_ID, T_ID, etc.
- There is no limit on how many tenant discriminator columns an application can define.
- Any name can be used for a discriminator column.
- Tenant discriminator column(s) must always be used with @Multitenant(SINGLE_TABLE). You cannot specify the tenant discriminator column(s) only.
- Generated schemas can include specified tenant discriminator columns.
- Tenant discriminator columns can be mapped or unmapped:
  - When a tenant discriminator column is mapped, its associated mapping attribute must be marked as read only. With this restriction in place, a tenant discriminator column cannot be part of the entity identifier; it can only be part of the primary key specification on the database.
- Both mapped and unmapped properties are used to form the additional criteria when issuing a SELECT query.

Using Single-Table Multi-Tenancy in an Inheritance Hierarchy

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

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

multi-tenancy at the root level, because you cannot isolate an entity to a single table to build only its type.

Examples

Table 2–97 shows a number of uses of tenant discriminator columns.

Example 2–97 Using @TenantDiscriminatorColumn Annotation

```
/** Single tenant discriminator column **/

@Entity
@Table(name = "CUSTOMER")
@Multitenant
@TenantDiscriminatorColumn(name = "TENANT", contextProperty = "multi-tenant.id")
public Customer() {
  ...
}


/** Multiple tenant discriminator columns using multiple tables **/

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


/** Tenant discriminator column mapped as part of the primary key on the database
**/

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


/** Mapped tenant discriminator column **/

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

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

Example 2–98 shows the same mappings, using the `<tenant-disciminator-column>` XML element in the `eclipselink-orm.xml` file.

Example 2–98 Using <tenant-discriminator-column> XML

```
<!-- Single tenant discriminator column -->

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



<!-- Multiple tenant discriminator columns using multiple tables -->

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



<!-- Tenant discriminator column mapped as part of the primary key on the database
-->

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



<!-- Mapped tenant discriminator column -->

<entity class="model.Player">
  <multi-tenant>
    <tenant-discriminator-column name="AGE" context-property="tenant.age"/>
  </multi-tenant>
```

```
  <table name="PLAYER"/>
  ...
  <attributes>
    <basic name="age" insertable="false" updatable="false">
      <column name="AGE"/>
    </basic>
    ...
  </attributes>
  ...
</entity>
```

See Also

- "@Multitenant" on page 82
- "@TenantDiscriminatorColumns" on page 162
- "@TenantTableDiscriminator" on page 164
-
- Multitenant Examples at http://wiki.eclipse.org/EclipseLink/Examples/JPA/Multitenant

@TenantDiscriminatorColumns

Specify multiple discriminator columns for single-table multitenancy by using the `@TenantDiscriminatorColumns` annotation to contain multiple `@TenantDiscriminatorColumn` annotations.

Annotation Elements

Table 2–62 describes this annotation's elements.

Table 2–62 @TenantDiscriminatorColumns Annotation Elements

Annotation Element	Description	Default
`TenantDiscriminatorColumn value`	(Optional) One or more `TenantDiscriminatorColumn` annotations.	none

Usage

You must use the `@TenantDiscriminatorColumns` annotation to contain multiple `@TenantDiscriminatorColumn` annotations. The `@TenantDiscriminatorColumns` annotation cannot be used alone, and multiple the `@TenantDiscriminatorColumn` annotations cannot be used alone, without `@TenantDiscriminatorColumns`.

Examples

```
@Entity
@Table(name = "EMPLOYEE")
@Multitenant(SINGLE_TABLE)
@TenantDiscriminatorColumns({
    @TenantDiscriminatorColumn(name = "TENANT_ID", contextProperty = "tenant-id)
    @TenantDiscriminatorColumn(name = "TENANT_CODE", contextProperty =
"tenant-code)})
```

See "@TenantDiscriminatorColumn" on page -156 for more examples of `@TenantDiscriminatorColumns`.

See Also

- "@Multitenant" on page -82
- "@TenantDiscriminatorColumn" on page -156
- "@TenantTableDiscriminator" on page -164

@TenantTableDiscriminator

Table-per-tenant multitenancy allows multiple tenants of an application to isolate their data in one or more tenant-specific tables. The tenant table discriminator specifies how to discriminate the tenant's tables from the other tenants' tables in a table-per-tenant multitenancy strategy.

Annotation Elements

Table 2–63 describes this annotation's elements.

Table 2–63 @TenantTableDiscriminator Annotation Elements

Annotation Element	Description	Default
`java.lang.String ContextProperty`	(Optional) Name of the context property to apply to as tenant table discriminator	`eclipselink-tenant.id`
`TenantTableDiscriminator type`	(Optional) Type of tenant table discriminator to use with the tables of the persistence unit.	`org.eclipse.persistence.annotations.TenantTableDiscriminatorType.SUFFIX`

Usage

In table-per-tenant multitenancy, tenants' tables can be in the same schema, using a prefix or suffix naming pattern to distinguish them; or they can be in separate schemas. The tenant table discriminator identifies whether to use the prefix or suffix naming pattern or to use a separate schema to identify and isolate the tenant's tables from other tenants' tables. The types are:

- Schema - Applies the tenant table discriminator as a schema to all multitenant tables. This strategy requires appropriate database provisioning.
- Suffix -Applies the tenant table discriminator as a suffix to all multitenant tables. This is the default strategy.
- Prefix - Applies the tenant table discriminator as a prefix to all multitenant tables.

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

For more information about using @TenantTableDiscriminator and table-per-tenant multitenancy, see "@Multitenant" on page -82.

Examples

The following example shows a `SCHEMA`-type table discriminator.

Example 2–99 @TenantTableDiscriminator Example 1

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

Example 2–100 Using <tenant-table-discriminator> XML

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

See Also

- "@Multitenant" on page 82
- "@TenantDiscriminatorColumn" on page 156
- "@TenantDiscriminatorColumns" on page 162
-
- Multitenant Examples at http://wiki.eclipse.org/EclipseLink/Examples/JPA/Multitenant

@TimeOfDay

Use `@TimeOfDay` to specify a specific time of day using a `Calendar` instance which is to be used within an `@OptimisticLocking` annotation.

Annotation Elements

Table 2–64 describes this annotation's elements.

Table 2–64 @TimeOfDay Annotation Elements

Annotation Element	Description	Default
`hour`	(Optional) Hour of the day	`0`
`millisecond`	(Optional) Millisecond of the day	`0`
`minute`	(Optional) Minute of the day	`0`
`second`	(Optional) Second of the day	`0`
`specified`	For internal use – do not modify	`true`

Examples

See "@Cache" on page -16 for examples of using `@TimeOfDay`.

See Also

For more information, see:

- "@ExistenceChecking" on page -56
- "@Cache" on page -16
- "How to Use the @TimeOfDay Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40TimeOfDay_Annotation
- "Cache Expiration and Invalidation" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Caching/Expiration

@Transformation

Use `@Transformation` with a Transformation mapping to define the transformation of database columns into attribute values (unless the Transformation mapping is write-only, in which case it should have a `@ReadTransformer` annotation).

Annotation Elements

Table 2–65 describes this annotation's elements.

Table 2–65 @Transformation Annotation Elements

Annotation Element	Description	Default
`fetch`	(Optional) Defines whether the value of the field or property should be lazily loaded or must be eagerly fetched. ■ The `EAGER` strategy is a requirement on the persistence provider runtime that the value must be eagerly fetched. ■ The `LAZY` strategy is a hint to the persistence provider runtime.	`EAGER`
`optional`	(Optional) A hint as to whether the value of the field or property may be `null`. It is disregarded for primitive types, which are considered non-optional.	`true`

Usage

Unless it's a read-only mapping, either `WriteTransformer` annotation or `WriteTransformers` annotation should be specified. Each `WriteTransformer` defines transformation of the attribute value to a single database column value (column is specified in the `WriteTransformer`).

Examples

Example 2–101 shows how to use the `@Transformation` annotation.

Example 2–101 Using @Transformation Annotation

```
@Transformation(fetch=FecthType.LAZY, optional="true")
@ReadTransformer(class=package.MyNormalHoursTransformer.class)
@WriteTranformers({
   @WriteTranformer(column=@Column(name="START_TIME"),
      method="getStartDate"),
   @WriteTranformer(column=@Column(name="END_TIME"),
      class=package.MyTimeTransformer.class)
})
@Mutable
@ReturnUpdate
@Access(AccessType.PROPERTY)
@AccessMethods(get="getNormalHours", set="setNormalHours")
@Properties({
   @Property(name="x", value="y")
})
```

Example 2–102 shows the same mapping, using the `<transformation>` XML element in the `eclipselink-orm.xml` file.

Example 2–102 Using <transformation> XML

```
<transformation name="normalHours" fetch="LAZY" optional="true">
     <read-transformer method="buildNormalHours"/>
```

```
    <write-transformer method="getStartTime">
            <column name="START_TIME"/>
    </write-transformer>
    <write-transformer class="package.MyTimeTransformer">
            <column name="END_TIME"/>
    </write-transformer>
    <mutable/>
    <return-update/>
    <access type="PROPERTY"/>
    <access-methods get="getNormalHours" set="setNormalHours"/>
    <properties>
       <property name="x" value="y"/>
    </properties>
</transformation>
```

See Also

For more information, see:

- "@WriteTransformer" on page -188
- "@ReadTransformer" on page -136
- "How to Use the @Transformation Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40Transformation_Annotation

@TypeConverter

Use `@TypeConverter` to modify data values during the reading and writing of a mapped attribute.

Annotation Elements

Table 2–66 describes this annotation's elements.

Table 2–66 @TypeConverter Annotation Elements

Annotation Element	Description	Default
`name`	(Required) The `String` name for your converter. This name must be unique across the persistence unit.	none
`dataType`	(Optional) The `type` stored in the database.	`void.class`[1]
`objectType`	(Optional) The `type` stored on the entity.	`void.class`[1]

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

Usage

Each `TypeConverter` must be uniquely named and can be defined at the class, field and property level and can be specified within an Entity, MappedSuperclass and Embeddable class. A `TypeConverter` is always specified by using an `@Convert` annotation

You can place a `@TypeConverter` on a `Basic`, `BasicMap` or `BasicCollection` mapping.

EclipseLink also includes `@ObjectTypeConverter` and `@StructConverter` converters.

Examples

Example 2–103 shows how to use the `@TypeConverter` annotation to convert the `Double` value stored in the database to a `Float` value stored in the entity.

Example 2–103 Using 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;
  }

...
}
```

See Also

For more information, see:

- "@Convert" on page -40
- "@TypeConverters" on page -172
- @TypeConverter
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Default_Conversions_and_Converters/TypeConverter
- "How to Use the @TypeConverter Annotation"
  http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40TypeConverter_Annotation
- "Default Conversions and Converters"
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Basic_Mappings/Default_Conversions_and_Converters

@TypeConverters

Use `@TypeConverters` to define multiple `TypeConverter` elements.

Annotation Elements

Table 2–67 describes this annotation's elements.

Table 2–67 @TypeConverters Annotation Elements

Annotation Element	Description	Default
`TypeConverter[]`	(Required) An array of type converter.	

Examples

Example 2–104 shows how to use this annotation.

Example 2–104 Using @TypeConverters Annotation

```
@Entity
@TypeConverters({

@TypeConverter(name="BigIntegerToString",dataType=String.class,objectType=BigInteg
er.class)
    })
public class Parameters implements Serializable {
    private static final long serialVersionUID = -1979843739878183696L;
    @Column(name="maxValue", nullable=false, length=512)
    @Convert("BigIntegerToString")
    private BigInteger maxValue;
...
}
```

Example 2–104 shows how to use the `<type-converters>` element in the `eclipselink-orm.xml` file.

Example 2–105 Using <type-converters> XML

```
<type-converters>
    <type-converter name="Long2String" data-type="String" object-type="Long"/>
    <type-converter name="String2String" data-type="String" object-type="String"/>
</type-converters>
<entity class="Employee">
...
</entity>
```

See Also

For more information, see:

- "@TypeConverter" on page -170
- "How to Use the @TypeConverter Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40TypeConverter_Annotation

@UuidGenerator

Use @UuidGenerator to defines a primary key generator that may be referenced by name when a generator element is specified for the `@GeneratedValue` annotation. A UUID (universally unique identifier) generator may be specified on the entity class or on the primary key field or property.

The generator name is global to the persistence unit (that is, across all generator types).

Annotation Elements

Table 2–68 describes this annotation's elements.

Table 2–68 @UuidGenerator Annotation Elements

Annotation Element	Description	Default
name	Name of the UUID generator, must be unique for the persistence unit	

Usage

To configure custom sequencing with a UUID generator, you must:

1. Implement a custom `Sequence`.
2. Use a `SessionCustomizer` to register the sequence.
3. Use the named sequence in the Entity.

Examples

In Example 2–106, Sequence is used to return random UUID value.

Example 2–106 Using @UuidGenerator Annotation

```
package eclipselink.example;

import java.util.UUID;
import java.util.Vector;

import org.eclipse.persistence.config.SessionCustomizer;
import org.eclipse.persistence.internal.databaseaccess.Accessor;
import org.eclipse.persistence.internal.sessions.AbstractSession;
import org.eclipse.persistence.sequencing.Sequence;
import org.eclipse.persistence.sessions.Session;

public class UUIDSequence extends Sequence implements SessionCustomizer {

  public UUIDSequence() {
    super();
  }

  public UUIDSequence(String name) {
    super(name);
  }

  @Override
  public Object getGeneratedValue(Accessor accessor,
      AbstractSession writeSession, String seqName) {
```

```
    return UUID.randomUUID().toString().toUpperCase();
  }

  @Override
  public Vector getGeneratedVector(Accessor accessor,
      AbstractSession writeSession, String seqName, int size) {
    return null;
  }

  @Override
  protected void onConnect() {
  }

  @Override
  protected void onDisconnect() {
  }

  @Override
  public boolean shouldAcquireValueAfterInsert() {
    return false;
  }

  @Override
  public boolean shouldOverrideExistingValue(String seqName,
      Object existingValue) {
    return ((String) existingValue).isEmpty();
  }

  @Override
  public boolean shouldUseTransaction() {
    return false;
  }

  @Override
  public boolean shouldUsePreallocation() {
    return false;
  }

  public void customize(Session session) throws Exception {
    UUIDSequence sequence = new UUIDSequence("system-uuid");

    session.getLogin().addSequence(sequence);
  }

}

...


properties.put(PersistenceUnitProperties.SESSION_CUSTOMIZER,
        "eclipselink.example.UUIDSequence");

...


@Id
@GeneratedValue(generator="system-uuid")
@Column(name="PROJ_ID")
private int id;
```

You can also specify the `SessionCustomizer` and configure the named sequence in your `eclipselink-orm.xml` file, as shown in Example 2–107.

Example 2–107 Using <generated-value> XML

```
<id name="id">
    <column name="PROJ_ID" />
    <generated-value generator="system-uuid"/>
</id>
```

You can also specify the named sequence at the persistence unit level (in the `persistence.xml` file) as shown in Example 2–108.

Example 2–108 Specifying Generator in persistence.xml

```
<property name="eclipselink.session.customizer"
value="eclipselink.example.UUIDSequence"/>
```

See Also

For more information, see:

- @Generated Value
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Entities/Ids/GeneratedValue
- "Entity Annotations" on page 2-2

@UnionPartitioning

Use `@UnionPartitioning` to send queries to all connection pools and then union the results. This can be used for queries or relationships that span partitions when partitioning is used, such as on a ManyToMany cross partition relationship.

Annotation Elements

Table 2–69 describes this annotation's elements.

Table 2–69 @UnionPartitioning Annotation Elements

Annotation Element	Description	Default
`name`	Name of the partition policy. Names must be unique for the persistence unit.	
`connectionPools`	List of connection pool names to load balance across	Defaults to all defined pools in the `ServerSession`
`replicateWrite`	Defines if write queries should be replicated. Writes are normally not replicated when unioning, but can be for ManyToMany relationships, when the join table needs to be replicated.	`false`

Usage

Partitioning can be enabled on an Entity, relationship, query, or session/persistence unit. Partition policies are globally named to allow reuse, the partitioning policy must also be set using the `@Partitioned` annotation to be used.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

If a transaction modifies data from multiple partitions, you should use JTA ensure proper two-phase commit of the data. You can also configure an exclusive connection in the EntityManager to ensure that only a single node is used for a single transaction.

Examples

See Example 2–75 on page -110 for an example of partitioning with EclipseLink.

See Also

For more information, see:

- "Data Partitioning" `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning`
- "@Partitioned" on page -110

@ValuePartition

Use `@ValuePartition` to represent a specific value partition that will be routed to a specific connection pool.

Annotation Elements

Table 2–70 describes this annotation's elements.

Table 2–70 @ValuePartition Annotation Elements

Annotation Element	Description	Default
`connectionPool`	The connection pool to which to route queries to for the `value`	
`value`	The `String` representation of the value	

Examples

Example 2–109 shows how to use the `@ValuePartition` and `@ValuePartitioning` annotations.

Example 2–109 Using @ValuePartition Annotation

```
@Entity
@Table(name = "PART_EMPLOYEE")
@IdClass(EmployeePK.class)
@ValuePartitioning(
  name="ValuePartitioningByLOCATION",
  partitionColumn=@Column(name="LOCATION"),
  unionUnpartitionableQueries=true,
  defaultConnectionPool="default",
  partitions={
    @ValuePartition(connectionPool="node2", value="Ottawa"),
    @ValuePartition(connectionPool="node3", value="Toronto")
  })
@Partitioned("ValuePartitioningByLOCATION")
public class Employee implements Serializable, Cloneable {
...
}
```

Example 2–110 shows how to use the `<partition>` element in the `eclipselink-orm.xml` file.

Example 2–110 Using <partition> XML

```
<entity name="Employee" class="Employee" access="FIELD">
  <table name="PART_EMPLOYEE"/>
  <id-class class="EmployeePK"/>
  <value-partitioning name="ValuePartitioningByLOCATION"
union-unpartitionable-queries="true" default-connection-pool="default">
    <partition-column name="LOCATION"/>
    <partition connection-pool="node2" value="Ottawa"/>
    <partition connection-pool="node3" value="Toronto"/>
  </value-partitioning>
<partitioned>ValuePartitioningByLOCATION</partitioned>
```

See Also

For more information, see:

- "@Partitioned" on page -110
- "@ValuePartitioning" on page -182

@ValuePartitioning

Use `@ValuePartitioning` to partition access to a database cluster by a field value from the object (such as the object's location or tenant). Each value is assigned a specific server. All write or read request for object's with that value are sent to the server. If a query does not include the field as a parameter, then it can either be sent to all server's and unioned, or left to the session's default behavior.

Annotation Elements

Table 2–71 describes this annotation's elements.

Table 2–71 @ValuePartitioning Annotation Elements

Annotation Element	Description	Default
`name`	(Required) Name of the partition policy. Names must be unique for the persistence unit.	
`partitionColumn`	(Required) The database column or query parameter to partition queries by This is the table column name, not the class attribute name. The column value must be included in the query and should normally be part of the object's ID. This can also be the name of a query parameter. If a query does not contain the field the query will not be partitioned.	
`partitions`	(Required) Store the value partitions. Each partition maps a value to a `connectionPool`.	
`defaultConnectionPool`	(Optional) The default connection pool is used for any unmapped values	
`partitionValueType`	(Optional) The `type` of the start and end values	`String`
`unionUnpartitionableQueries`	(Optional) Defines if queries that do not contain the partition field should be sent to every database and have the result unioned.	`false`

Usage

Partitioning can be enabled on an Entity, relationship, query, or session/persistence unit. Partition policies are globally named to allow reuse, the partitioning policy must also be set using the `@Partitioned` annotation to be used.

The persistence unit properties support adding named connection pools in addition to the existing configuration for read/write/sequence. A named connection pool must be defined for each node in the database cluster.

If a transaction modifies data from multiple partitions, you should use JTA ensure proper two-phase commit of the data. You can also configure an exclusive connection in the EntityManager to ensure that only a single node is used for a single transaction.

Examples

See Example 2–75 on page -110 for an example of partitioning with EclipseLink.

See Also

For more information, see:

- "Data Partitioning" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Data_Partitioning
- "@Partitioned" on page -110

@VariableOneToOne

Use `@VariableOneToOne` to represent a pointer references between a java object and an implementer of an interface. This mapping is usually represented by a single pointer (stored in an instance variable) between the source and target objects. In the relational database tables, these mappings are normally implemented using a foreign key and a type code.

Annotation Elements

Table 2–72 describes this annotation's elements.

Table 2–72 @VariableOneToOne Annotation Elements

Annotation Element	Description	Default
`CascadeType`	(Optional) Array of operations that must be cascaded to the target of the association.	
`DiscriminatorClasses`	(Optional) Array of discriminator types that can be used with this mapping.	If none are specified, EclipseLink adds entities within the persistence unit that implement the target interface. If `DiscriminatorColumn` is `STRING`, EclipseLink uses `Entity.name()`. If `DiscriminatorColumn` is `CHAR`, EclipseLink uses the first letter of the entity class. If `DiscriminatorColumn` is `INTEGER`, EclipseLink uses the next integer after the highest integer explicitly stated.
`DiscriminatorColumn`	(Optional) The discriminator column that contains the type identifiers.	`DTYPE`
`FetchType`	(Optional) Specify how the value of the field or property should be loaded: ■ **Eager**: Requires that the persistence provider runtime must eagerly fetch the value ■ **Lazy**: Hints that the persistence provider should lazily load the value	`Eager`
`Optional`	(Optional) Specify if the association is optional.	
`OrphanRemoval`	(Optional) Specify if interface class that is the target of this mapping.	
`TargetInterface`	(Optional) The interface class that is the target of this mapping.	If none is specified, EclipseLink will infer the interface class based on the type of object being referenced.

Usage

You can specify `@VariableOneToOne` on an Entity, MappedSuperclass, or Embeddable class.

Examples

Example 2–111 shows how to use the `@VariableOneToOne` annotation.

Example 2–111 Using @VariableOneToOne Annotation

```
@VariableOneToOne(
    cascade={ALL},
    fetch=LAZY,
    discriminatorColumn=@DiscriminatorColumn(name="CONTACT_TYPE"),
    discriminatorClasses={
        @DiscriminatorClass(discriminator="E", value="Email.class"),
        @DiscriminatorClass(discriminator="P", value="Phone.class")
    }
}
@JoinColumn(name="CONTACT_ID", referencedColumnName="C_ID")
@PrivateOwned
@JoinFetch(INNER)
public Contact getContact() {
    return contact;
}
```

Example 2–112 shows the same mapping using the `<variable-one-to-one>` XML element in the `eclipselink-orm.xml` file.

Example 2–112 Using <variable-one-to-one> XML

```
<variable-one-to-one name="contact" fetch="LAZY">
    <cascade>
        <cascade-all/>
    </cascade>
    <discriminator-column name="CONTACT_TYPE"/>
    <discriminator-class discriminator="E" value="Email.class"/>
    <discriminator-class discriminator="P" value="Phone.class"/>
    <join-column name="CONTACT_ID" referenced-column-name="C_ID"/>
    <private-owned/>
    <join-fetch>INNER</join-fetch>
</variable-one-to-one>
```

See Also

For more information, see:

- "@DiscriminatorClass" on page -52
- "How to Use the @VariableOneToOne Annotation" http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_%28ELUG%29#How_to_Use_the_.40VariableOneToOne_Annotation
- "Variable One-To-One Mapping" http://wiki.eclipse.org/Introduction_to_Relational_Mappings_%28ELUG%29#Variable_One-to-One_Mapping

@VirtualAccessMethods

Use `@VirtualAccessMethods` to specify that a specific class contains virtual methods.

Annotation Elements

Table 2–73 describes this annotation's elements.

Table 2–73 @VirtualAccessMethods Annotation Elements

Annotation Element	Description	Default
`get`	(Optional) Name of the `getter` method to use for the virtual property This method must take a single `java.lang.String` parameter and return a `java.lang.Object`. If `get` is specified, you must also specify `set`.	`get`
set	(Optional) Name of the `setter` method to use for the virtual property This method must take a `java.lang.String` parameter and a `java.lang.Object` parameter. If `set` is specified, you must also specify `get`.	`set`

Usage

Use the `@VirtualAccessMethods` annotation to define access methods for mappings with in which `accessType`=VIRTUAL.

Examples

Table 2–73 shows an entity using property access.

Example 2–113 Using @VirtualAccessMethods Annotation

```
@Entity
  @VirtualAccessMethods
  public class Customer{

    @Id
    private int id;
...

    @Transient
    private Map<String, Object> extensions;

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

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

In addition to using the `@VirtualAccessMethods` annotation, you can use the `<access>` and `<access-method>` elements in your `eclipselink-orm.xml` file, as shown in Example 2–114.

Example 2–114 Using <access> and <access-methods> XML

```
<access>VIRTUAL</access>
```

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

See Also

For more information, see:

- "Extensible Entities" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Extensible_Entities

@WriteTransformer

Use `@WriteTransformer` on a `TranformationMapping` to transform a single attribute value to a single database column value. Use the `@WriteTransformers` annotation to wrap multiple transformations.

Annotation Elements

Table 2–74 describes this annotation's elements.

Table 2–74 @WriteTransformer Annotation Elements

Annotation Element	Description	Default
`column`	(Optional) The column into which the value should be written. If a single `WriteTransfomer` annotates an attribute, the attribute's name will be used as the column name.	`@javax.persistence.Column`
`method`	(Optional) The `String` method name that the mapped class must have. This method returns the value to be written into the database column. **Note**: To support DDL generation and returning policy, the method should be defined to return a particular type, not just an `Object`. For example: `public Time getStartTime()` The method may require `@Transient` to avoid being mapped as a `Basic` by default.	
`transformerClass`	(Optional) User-defined class that implements the `FieldTransformer` interface. This will instantiate the class and use its `buildFieldValue` method to create the value to be written into the database column. **Note**: To support DDL generation and returningpolicy, the method `buildFieldValue` in the class should be defined to return the relevant Java type, not just `Object` as defined in the interface. For example: `public Time buildFieldValue(Object instance, String fieldName, Session session).`	`void.class`

Note: You must specify either `transformerClass` *or* `method` , but not both.

Usage

You cannot define a `@WriteTransformer` for a read-only mapping.

Unless the `TransformationMapping` is write-only, it should include a `ReadTransformer` that defines the transformation of the database column values into attribute values.

Configuring Field Transformer Associations

Using a `FieldTransformer` is non-intrusive; your domain object does not need to implement an EclipseLink interface or provide a special transformation method.

You can configure a method-based field transformer using `AbstractTransformationMapping` method `addFieldTransformation`, passing in the name of the database field and the name of the domain object method to use.

You can configure a class-based field transformer using `AbstractTransformationMapping` method `addFieldTransformer`, passing in the name of the database field and an instance of `org.eclipse.persistence.mappings.Transfomers.FieldTransformer`.

A convenient way to create a `FieldTransformer` is to extend `FieldTransformerAdapter`.

Examples

See "Using @Transformation Annotation" on page -168 for an example of how to use the `@WriteTransformer annotation` with a Transformation mapping.

See Also

For more information, see:

- "@WriteTransformers" on page -190
- "@Transformation" on page -168.
- http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Print_Version#How_to_Use_the_.40Transformation_Annotation

@WriteTransformers

Use `@WriteTransformer` on a `TranformationMapping` to transform a single attribute value to a single database column value. Use the `@WriteTransformers` annotation to wrap multiple transformations.

Annotation Elements

Table 2–75 describes this annotation's elements.

Table 2–75 @WriteTransformers Annotation Elements

Annotation Element	Description	Default
`WriteTransformer`	An array of `WriteTransformer`	

Usage

You cannot use `@WriteTransformer`s for a read-only mapping.

Examples

See "Using @Transformation Annotation" on page -168 for an example of how to use the `@WriteTransformer annotation` with a Transformation mapping.

See Also

For more information, see:

- "@WriteTransformer" on page -188.
- "@Transformation" on page -168.
- http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Print_Version#How_to_Use_the_.40Transformation_Annotation

3

Java Persistence Query Language Extensions

EclipseLink provides many extensions to the standard JPA Java Persistence Query Language (JPQL). These extensions, referred to as the EclipseLink Query Language (EQL), provide access to additional database features many of which are part of standard SQL, provide access to native database features and functions, and provide access to EclipseLink specific features.

This chapter contains the following sections:

- Special Operators
- EclipseLink Query Language

For more information on JQPL, see:

- "Query Language" in the JPA Specification (`http://jcp.org/en/jsr/detail?id=317`)
- "The Java Persistence Query Language" in *The Java EE 6 Tutorial* (`http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html`)

Special Operators

EclipseLink defines the following operators to perform database operations that would not be possible in standard JPQL:

- COLUMN
- FUNCTION
- OPERATOR
- SQL

EclipseLink Query Language

The following lists the EQL extensions to JPQL:

- CAST
- EXCEPT
- EXTRACT
- INTERSECT
- ON

- REGEXP
- TABLE
- TREAT
- UNION

CAST

Use `CAST` to convert a value to a specific database type.

Usage

The `CAST` function is database independent, but requires database support.

Examples

Example 3–1 shows how to use this JPQL extension.

Example 3–1 Using CAST EQL

```
CAST(e.salary NUMERIC(10,2))
```

See Also

For more information, see:

- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
-

COLUMN

Use `COLUMN` to access to unmapped columns in an object's table.

Usage

You can use `COLUMN` to access foreign key columns, inheritance discriminators, or primitive columns (such as `ROWID`). You can also use `COLUMN` in JPQL fragments inside the `@AdditionalCriteria` annotation.

Examples

Example 3–2 shows how to use the `COLUMN` EQL.

Example 3–2 Using COLUMN EQL

```
SELECT e FROM Employee e WHERE COLUMN('MANAGER_ID', e) = :id
```

In Example 3–3, uses `COLUMN` EQL access a primitive column (`ROWID`).

Example 3–3 Using COLUMN with a Primitive Column

```
SELECT e FROM Employee e WHERE COLUMN('ROWID', e) = :id
```

See Also

For more information, see:

- "@AdditionalCriteria" on page -8
- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

EXCEPT

When performing multiple queries, use `EXCEPT` to remove the results of a second query from the results of a first query.

Usage

The `EXCEPT` function is database independent, but requires database support.

Examples

Example 3–4 shows how to use this JPQL extension.

Example 3–4 Using EXCEPT EQL

```
SELECT e FROM Employee e
EXCEPT SELECT e FROM Employee e WHERE e.salary > e.manager.salary
```

See Also

For more information, see:

- "UNION" on page -30
- "INTERSECT" on page -14
- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

EXTRACT

Use `EXTRACT` to retrieve the date portion of a date/time value.

Usage

The `EXTRACT` function is database independent, but requires database support

Examples

Example 3–5 shows how to use this JPQL extension.

Example 3–5 Using EXTRACT EQL

```
EXTRACT(YEAR, e.startDate)
```

See Also

For more information, see:

- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
-

FUNCTION

Use FUNCTION (formerly FUNC) to call database specific functions from JPQL

Usage

You can use FUNCTION to call database functions that are not supported directly in JPQL and to call user or library specific functions.

> **Note:** FUNCTION is database specific – it does not translate the function call in any way to support different databases as other JPQL functions do.

Use FUNCTION to call functions with normal syntax. Functions that require special syntax cannot be called with FUNCTION. Instead, use OPERATOR

Examples

Example 3–6 shows how to use this JPQL extension.

Example 3–6 Using FUNCTION EQL

```
SELECT p FROM Phone p WHERE FUNCTION('TO_NUMBER', e.areaCode) > 613

SELECT FUNCTION('YEAR', e.startDate) AS year, COUNT(e) FROM Employee e GROUP BY
year
```

Example 3–7 shows how to use FUNCTION with Oracle Spatial queries

Example 3–7 Using FUNCTION EQL Oracle Spatial examples

```
SELECT a FROM Asset a, Geography geo WHERE geo.id = :id AND a.id IN :id_list AND
FUNCTION('ST_INTERSECTS', a.geometry, geo.geometry) = 'TRUE'


SELECT s FROM SimpleSpatial s WHERE FUNCTION('MDSYS.SDO_RELATE', s.jGeometry,
:otherGeometry, :params) = 'TRUE' ORDER BY s.id ASC
```

See Also

For more information, see:

- "OPERATOR" on page -18
- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

INTERSECT

When performing multiple queries, use `INTERSECT` to return only results that are found in both queries.

Examples

Example 3–8 shows how to use this JPQL extension.

Example 3–8 Using INTERSECT EQL

```
SELECT e FROM Employee e JOIN e.phones p WHERE p.areaCode = :areaCode1
INTERSECT SELECT e FROM Employee e JOIN e.phones p WHERE p.areaCode = :areaCode2
```

See Also

For more information, see:

- "UNION" on page -30
- "EXCEPT" on page -8
- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

ON

Use the `ON` clause to append additional conditions to a `JOIN` condition, such as for outer joins.

Usage

EclipseLink supports using the `ON` clause between two root level objects.

Examples

Example 3–9 shows how to use this JPQL extension.

Example 3–9 Using ON Clause EQ

```
SELECT e FROM Employee e LEFT JOIN e.address ON a.city = :city


SELECT e FROM Employee e LEFT JOIN MailingAddress a ON e.address = a.address
```

See Also

For more information, see:

- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
-

OPERATOR

Use `OPERATION` to call any EclipseLink operator.

Usage

EclipseLink supports many database functions using standard operator names that are translated to different databases. EclipseLink operators are supported on any database that has an equivalent function (or set of functions). Use the EclipseLink `ExpressionOperator` class to define a custom operator or allow `DatabasePlatform` to override an operator..

`OPERATOR` is similar to `FUNCTION`, but allows the function to be database independent, and you can call functions that require special syntax.

The supported EclipseLink operators include:

- Abs
- ToUpperCase
- ToLowerCase
- Chr
- Concat
- Coalesce
- Case
- HexToRaw
- Initcap
- Instring
- Soundex
- LeftPad
- LeftTrim
- RightPad
- RightTrim
- Substring
- Translate
- Ascii
- Length
- CharIndex
- Cast
- Extract
- CharLength
- Difference
- Reverse
- Replicate

- Right
- Locate
- ToNumber
- ToChar
- AddMonths
- DateToString
- MonthsBetween
- NextDay
- RoundDate
- AddDate
- DateName
- DatePart
- DateDifference
- TruncateDate
- NewTime
- Nvl
- NewTime
- Ceil
- Cos
- Cosh
- Acos
- Asin
- Atan
- Exp
- Sqrt
- Floor
- Ln
- Log
- Mod
- Power
- Round
- Sign
- Sin
- Sinh
- Tan
- Tanh
- Trunc

- Greatest
- Least
- Add
- Subtract
- Divide
- Multiply
- Atan2
- Cot
- Deref
- Ref
- RefToHex
- Value
- ExtractXml
- ExtractValue
- ExistsNode
- GetStringVal
- GetNumberVal
- IsFragment
- SDO_WITHIN_DISTANCE
- SDO_RELATE
- SDO_FILTER
- SDO_NN
- NullIf

Examples

Example 3–10 shows how to use this JPQL extension.

Example 3–10 Using OPERATOR EQL

```
SELECT e FROM Employee e WHERE OPERATOR('ExtractXml', e.resume,
'@years-experience') > 10
```

See Also

For more information, see:

- "FUNCTION" on page -12
- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

REGEXP

Use `REGEXP` to determine if a string matches a regular expression.

Usage

To use the `REGEXP` function, your database must support regular expressions.

Examples

Example 3–11 shows how to use this JPQL extension.

Example 3–11 Using REGEXP EQL

```
e.lastName REGEXP '^Dr\.*'
```

See Also

For more information, see:

- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
-

SQL

Use `SQL` to integrate SQL within a JPQL statement. This provides an alternative to using native SQL queries simply because the query may require a function not supported in JPQL.

Usage

The `SQL` function includes both the SQL string (to inline into the JPQL statement) and the arguments to translate into the SQL string. Use a question mark character (**?**) to define parameters within the SQL that are translated from the SQL function arguments.

You can use `SQL` to call database functions with non standard syntax, embed SQL literals, and perform any other SQL operations within JPQL. With SQL, you can still use JPQL for the query.

Examples

Example 3–12 shows how to use this JPQL extension.

Example 3–12 Using SQL EQ

```
SELECT p FROM Phone p WHERE SQL('CAST(? AS CHAR(3))', e.areaCode) = '613'


SELECT SQL('EXTRACT(YEAR FROM ?)', e.startDate) AS year, COUNT(e) FROM Employee e
GROUP BY year


SELECT e FROM Employee e ORDER BY SQL('? NULLS FIRST', e.startDate)


SELECT e FROM Employee e WHERE e.startDate = SQL('(SELECT SYSDATE FROM DUAL)')
```

See Also

For more information, see:

- "JPQL"
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
-

TABLE

Use `TABLE` to access unmapped tables.

Usage

With the `TABLE` function, you use join, collection, history, auditing, or system tables in a JPQL query.

Examples

Example 3–13 shows how to use an **audit** table (unmapped) within a `SELECT` statement.

Example 3–13 Using TABLE EQL

```
SELECT e, a.LAST_UPDATE_USER FROM Employee e, TABLE('AUDIT') a WHERE a.TABLE =
'EMPLOYEE' AND a.ROWID = COLUMN('ROWID', e)
```

See Also

For more information, see:

- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
-

TREAT

Use `TREAT` to cast an object as its subclass value (that is, downcast related entities with inheritance).

Examples

Example 3–14 shows how to use this JPQL extension.

Example 3–14 Using TREAT EQL

```
SELECT e FROM Employee JOIN TREAT(e.projects AS LargeProject) p WHERE p.budget >
1000000
```

See Also

For more information, see:

- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
-

UNION

Use `UNION` to combine the results of two queries into a single query.

Usage

With `UNION`, the unique results from both queries will be returned. If you include the `ALL` option, the results found in both queries will be duplicated.

Examples

Example 3–15 shows how to use this JPQL extension.

Example 3–15 Using UNION EQL

```
SELECT MAX(e.salary) FROM Employee e WHERE e.address.city = :city1
UNION SELECT MAX(e.salary) FROM Employee e WHERE e.address.city = :city2
```

See Also

For more information, see:

- "EXCEPT" on page -8
- "INTERSECT" on page -14
- "JPQL" http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

4

JPA Query Customization Extensions

You can specify EclipseLink query hints (JPA query extensions) by:

- Using the `@QueryHint` annotation
- Including the hints in the `orm.xml` or `eclipselink-orm.xml` file
- Using the `setHint()` method when executing a named or dynamic query (JPQL or Criteria)

EclipseLink supplies the following query hints:

- batch
- cache-usage
- jdbc.bind-parameters
- jdbc.fetch-size
- jdbc.max-rows
- jdbc.result-collection-type
- jdbc.timeout
- join-fetch
- maintain-cache
- pessimistic-lock
- query-type
- read-only
- refresh
- refresh.cascade

All EclipseLink query hints are defined in the `QueryHints` class in the `org.eclipse.persistence.config package`. When you set a hint, you can set the value using the public static final field in the appropriate configuration class in `org.eclipse.persistence.config` package, including the following:

- `HintValues`
- `CacheUsage`
- `PessimisticLock`
- `QueryType`

For more information, see Section 10.3.1 "NamedQuery Annotation" in the JPA Specification (`http://jcp.org/en/jsr/detail?id=317`).

batch

Use `eclipselink.batch` to supply 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.

Values

This query hint accepts a single-valued, relationship path expression.

Usage

Using the `eclipselink.batch` hint is more efficient than joining, because EclipseLink avoids reading duplicate data.

You can only batch queries that have a single object in the 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, use `e.manager.address`.

Examples

Example 4–1 shows how to use this hint in a JPA query.

Example 4–1 Using batch in a JPA Query

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint("eclipselink.batch", "e.address");
```

Example 4–2 shows how to use this hint with the `@QueryHint` annotation.

Example 4–2 Using batch in a @QueryHint Annotation

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 @QueryHint(name=QueryHints.BATCH, value="e.address");
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- "join-fetch" on page -16

cache-usage

Use `eclipselink.cache-usage` to specify how the query should interact with the EclipseLink cache.

Values

Table 4–1 describes this query hint's valid values.

Table 4–1 Valid Values for org.eclipse.persistence.config.CacheUsage

Value	Description
`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.
`UseEntityDefault`	(Default) 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.

Usage

EclipseLink JPA uses a shared cache assessed across the entire persistence unit. After completing an operation in a particular persistence context, EclipseLink merges the results into the shared cache, so that other persistence contexts can use the results *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 consistently maintained with the cache.

Examples

Example 4–3 shows how to use this hint in a JPA query.

Example 4–3 Using cache-usage in a JPA Query

```
import org.eclipse.persistence.config.CacheUsage;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.CACHE_USAGE, CacheUsage.CheckCacheOnly);
```

Example 4–4 shows how to use this hint with the `@QueryHint` annotation.

Example 4–4 Using cache-usage in a @QueryHint Annotation

```
import org.eclipse.persistence.config.CacheUsage;
 import org.eclipse.persistence.config.TargetDatabase;
 @QueryHint(name=QueryHints.CACHE_USAGE, value=CacheUsage.CheckCacheOnly);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints

jdbc.bind-parameters

Use `eclipselink.jdbc.bind-parameters` to specify if the query uses parameter binding (parameterized SQL).

Values

Table 4–2 describes this query hint's valid values.

Table 4–2 Valid Values for org.eclipse.persistence.config.HintValues

Value	Description
`TRUE`	Bind all parameters.
`FALSE`	Do not bind all parameters.
`PERSISTENCE_UNIT_DEFAULT`	(Default) Use the parameter binding setting made in your EclipseLink session's database login, which is true by default.

Usage

By default, EclipseLink enables parameter binding and statement caching. This causes EclipseLink to use a prepared statement, binding all SQL parameters and caching the prepared statement. When you re-execute this query, you avoid the SQL preparation, which improves performance.

You can also configure parameter binding for the persistence unit in the `persistence.xml` file (when used in a Java SE environment).

Examples

Example 4–5 shows how to use this hint in a JPA query.

Example 4–5 Using bind-parameters in a JPA Query

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.BIND_PARAMETERS, HintValues.TRUE);
```

Example 4–6 shows how to use this hint with the `@QueryHint` annotation.

Example 4–6 Using bind-parameters in a @QueryHint Annotation

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.TargetDatabase;
 @QueryHint(name=QueryHints.BIND_PARAMETERS, value=HintValues.TRUE);
```

Example 4–7 shows how to configure parameter binding in the persistence unit `persistence.xml` file.

Example 4–7 Specifying Parameter Binding Persistence Unit Property

```
<property name="eclipselink.jdbc.bind-parameters" value="false"/>
```

Or by importing a `property` map:

```
import org.eclipse.persistence.config.PersistenceUnitProperties;
propertiesMap.put(PersistenceUnitProperties.NATIVE_SQL, "true");
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- "How to Use Parameterized SQL and Statement Caching in a DatabaseQuery"
  http://wiki.eclipse.org/Using_Basic_Query_API_%28ELUG%29#How_to_Use_Parameterized_SQL_and_Statement_Caching_in_a_DatabaseQuery
- "How to Use Parameterized SQL (Parameter Binding) and Prepared Statement Caching for Optimization"
  http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#How_to_Use_Parameterized_SQL_.28Parameter_Binding.29_and_Prepared_Statement_Caching_for_Optimization

jdbc.fetch-size

Use `eclipselink.jdbc.fetch-size` to specify the number of rows to be fetched from the database when additional rows are needed.

Note: This property requires JDBC driver support.

Values

Table 4–3 describes this query hint's valid values.

Table 4–3 Valid Values for eclipselink.jdbc.fetch-size

Value	Description
from `0` to `Integer.MAX_VALUE`	(Default = `0`) As a `String`, depending on your JDBC driver. If 0, the JDBC driver default will be used.

Usage

For 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.

By default, most JDBC drivers use 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.

If you are unsure of the result set size, incorrectly setting a fetch size too large or too small can decrease performance.

Examples

Example 4–8 shows how to use this hint in a JPA query.

Example 4–8 Using jdbc.fetch-size in a JPA Query

Example 4–9 shows how to use this hint with the `@QueryHint` annotation.

Example 4–9 Using jdbc.fetch-size in a @QueryHint Annotation

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- "How to Use JDBC Fetch Size for Optimization"
  http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#How_to_Use_JDBC_Fetch_Size_for_Optimization

jdbc.max-rows

Use `eclipselink.jdbc.max-rows` to specify the maximum number of rows to be returned. If the query returns more rows than specified, the trailing rows will not be returned.

Values

Table 4–4 describes this query hint's valid values.

Table 4–4 Valid Values for eclipselink.jdbc.max-rows

Value	Description
`Int` or `String` (that can be parsed to `Int` values)	Configures the JDBC maximum number of rows.

Usage

This hint is similar to JPQL `setMaxResults()`, but can be specified within the metadata for `NamedQueries`.

Examples

Example 4–10 shows how to use this hint in a JPA query.

Example 4–10 Using jdbc.max-rows in a JPA Query

Example 4–11 shows how to use this hint with the `@QueryHint` annotation.

Example 4–11 Using jdbc.max-rows in a @QueryHint Annotation

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints`
-

jdbc.result-collection-type

Usage

Examples

See Also

For more information, see:

-
-

jdbc.timeout

Use `eclipselink.jdbc.timeout` to specify number of seconds EclipseLink will wait (time out) for a query result, before throwing a `DatabaseExcpetion`.

Note: This property requires JDBC driver support.

Values

Table 4–5 describes this query hint's valid values.

Table 4–5 Valid Values for eclipselink.jdbc.timeout

Value	Description
from `0` to `Integer.MAX_VALUE`	(Default = `0`) As a `String`, depending on your JDBC driver. If 0, EclipseLink will never time out waiting for a query.

Usage

Examples

Example 4–12 shows how to use this hint in a JPA query.

Example 4–12 Using cache-usage in a JPA Query

```
import org.eclipse.persistence.config.CacheUsage;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.CACHE_USAGE, CacheUsage.CheckCacheOnly);
```

Example 4–13 shows how to use this hint with the `@QueryHint` annotation.

Example 4–13 Using cache-usage in a @QueryHint Annotation

```
import org.eclipse.persistence.config.CacheUsage;
 import org.eclipse.persistence.config.TargetDatabase;
 @QueryHint(name=QueryHints.CACHE_USAGE, value=CacheUsage.CheckCacheOnly);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
-

join-fetch

Use `eclipselink.join-fetch hint` to join attributes in a query.

> **Note:** Use *dot notation* to access nested attributes. For example, to batch-read an employee's manager's address, use `e.manager.address`.

Values

Table 4–6 describes this query hint's valid values.

Table 4–6 Valid Values for eclipselink.join-fetch hint

Value
A relationship path expression

Usage

This hint is similar to `eclipselink.batch`. Subsequent queries of related objects can be optimized in batches instead of being retrieved in one large joined read

The `eclipselink.join-fetch` hint differs from JPQL joining in that it allows multilevel fetch joins.

Examples

Example 4–14 shows how to use this hint in a JPA query.

Example 4–14 Using join-fetch in a JPA Query

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint("eclipselink.join-fetch", "e.address");
```

Example 4–15 shows how to use this hint with the `@QueryHint` annotation.

Example 4–15 Using join-fetch in a @QueryHint Annotation

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 @QueryHint(name=QueryHints.FETCH, value="e.address");
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- EclipseLink Examples
  http://wiki.eclipse.org/EclipseLink/Examples/JPA/QueryOptimization

- "Optimizing Queries" http://wiki.eclipse.org/Optimizing_the_EclipseLink_Application_%28ELUG%29#Optimizing_Queries
- "Fetch Joins" in the JPA Specification (http://jcp.org/en/jsr/detail?id=317)
- "batch" on page -2

maintain-cache

Use `eclipselink.maintain-cache` to controls whether or not query results are cached in the session cache

Values

Table 4–7 describes this query hint's valid values.

Table 4–7 Valid Values for org.eclipselink.maintain-cache

Value	Description
`TRUE`	Maintain cache.
`FALSE`	(Default) Do not maintain cache.

Usage

The `eclipselink.maintain-cache` hint provides a way to query the current database contents *without affecting the current persistence context*. It configures the query to return un-managed instances so any updates to entities queried using this hint would have to be merged into the persistence context.

Examples

Example 4–16 shows how to use this hint in a JPA query.

Example 4–16 Using maintain-cache in a JPA Query

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.MAINTAIN_CACHE, HintValues.FALSE);
```

Example 4–17 shows how to use this hint with the `@QueryHint` annotation.

Example 4–17 Using maintain-cache in a @QueryHint Annotation

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 @QueryHint(name=QueryHints.MAINTAIN_CACHE, value=HintValues.FALSE);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints

pessimistic-lock

Use `eclipselink.pessimistic-lock` to specify if EclipseLink uses pessimistic locking.

Values

Table 4–8 describes this query hint's valid values.

Table 4–8 Valid Values for org.eclipse.persistence.config.PessimisticLock

Value	Description
`NoLock`	(Default) Do not use pessimistic locking.
`Lock`	EclipseLink issues `SELECT .... FOR UPDATE` statements.
`LockNoWait`	EclipseLink issues `SELECT .... FOR UPDATE NO WAIT` statements.

Usage

The primary advantage of using pessimistic locking is that you are assured, once the lock is obtained, of a successful edit. This is desirable in highly concurrent applications in which optimistic locking may cause too many optimistic locking errors.

One drawback of pessimistic locking is that it requires additional database resources, requiring the database transaction and connection to be maintained for the duration of the edit. Pessimistic locking may also cause deadlocks and lead to concurrency issues.

Examples

Example 4–18 shows how to use this hint in a JPA query.

Example 4–18 Using pessimistic-lock in a JPA Query

```
import org.eclipse.persistence.config.PessimisticLock;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.PESSIMISTIC_LOCK, PessimisticLock.LockNoWait);
```

Example 4–19 shows how to use this hint with the `@QueryHint` annotation.

Example 4–19 Using pessimistic-lock in a @QueryHint Annotation

```
import org.eclipse.persistence.config.PessimisticLock;
 import org.eclipse.persistence.config.QueryHints;
 @QueryHint(name=QueryHints.PESSIMISTIC_LOCK, value=PessimisticLock.LockNoWait);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- EclipseLink Examples
  http://wiki.eclipse.org/EclipseLink/Examples/JPA/PessimisticLocking

- "Locking"
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Mapping/Locking

query-type

Use `eclipselink.query-type` to specify which EclipseLink query type to use for the query.

Values

Table 4–9 describes this query hint's valid values.

Table 4–9 Valid Values for org.eclipse.persistence.config.QueryType

Value	Description
`Auto`	(Default = `0`) EclipseLink chooses the type of query.
`ReadAll`	Use a `ReadAllQuery`.
`ReadObject`	Use a `ReadObjectQuery`.
`Report`	Use a `ReportQuery`.

Usage

By default, EclipseLink uses `org.eclipse.persistence.queries.ReportQuery` or `org.eclipse.persistence.queries.ReadAllQuery` for most JPQL queries. Use the `eclipselink.query-type` hint lets to specify another query type, such as `org.eclipse.persistence.queries.ReadObjectQuery` for queries that will return a single object.

Examples

Example 4–20 shows how to use this hint in a JPA query.

Example 4–20 Using query-type in a JPA Query

```
import org.eclipse.persistence.config.QueryType;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.QUERY_TYPE, QueryType.ReadObject);
```

Example 4–21 shows how to use this hint with the `@QueryHint` annotation.

Example 4–21 Using query-type in a @QueryHint Annotation

```
import org.eclipse.persistence.config.QueryType;
 import org.eclipse.persistence.config.TargetDatabase;
 @QueryHint(name=QueryHints.QUERY_TYPE, value=QueryType.ReadObject);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints

read-only

Use `eclipselink.read-only` to retrieve read-only results back from a query.

Values

Table 4–10 describes this query hint's valid values.

Table 4–10 Valid Values for org.eclipse.persistence.config.HintValues

Value	Description
`TRUE`	Retrieve read-only results from the query.
`FALSE`	(Default) Do not retrieve read-only results from the query.

Usage

For non-transactional read operations, if 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.

Examples

Example 4–22 shows how to use this hint in a JPA query.

Example 4–22 Using read-only in a JPA Query

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.READ_ONLY, HintValues.TRUE);
```

Example 4–23 shows how to use this hint with the `@QueryHint` annotation.

Example 4–23 Using read-only in a @QueryHint Annotation

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 @QueryHint(name=QueryHints.READ_ONLY, value=HintValues.TRUE);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- "Using Read-Only Queries"
  http://wiki.eclipse.org/Using_Advanced_Query_API_%28ELUG%29#Using_Read-Only_Queries

refresh

Use `eclipselink.refresh` to specify whether or not to update the EclipseLink session cache with objects returned by the query.

Values

Table 4–11 describes this query hint's valid values.

Table 4–11 Valid Values for eclipselink.refresh

Value	Description
`FALSE`	Refresh the cache.
`FALSE`	(Default) Do not refresh the cache.

Usage

Examples

Example 4–24 shows how to use this hint in a JPA query.

Example 4–24 Using refresh in a JPA Query

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.REFRESH, HintValues.TRUE);
```

Example 4–25 shows how to use this hint with the `@QueryHint` annotation.

Example 4–25 Using refresh in a @QueryHint Annotation

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 @QueryHint(name=QueryHints.REFRESH, value=HintValues.TRUE);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- "refresh.cascade" on page -28

refresh.cascade

Use `eclipselink.refresh.cascade` to specify if a refresh query should cascade the refresh to relationships.

Values

Table 4–12 describes this query hint's valid values.

Table 4–12 Valid Values for eclipselink.refresh.cascade

Value	Description
`CascadeAllParts`	Cascade to all associations.
CascadeByMapping	Cascade by mapping metadata.
CascadePrivateParts	Cascade to privately-owned relationships.
NoCascade	Do not cascade.

Usage

You should also use a `refresh` hint in order to cause the refresh.

Examples

Example 4–26 shows how to use this hint in a JPA query.

Example 4–26 Using refresh.cascade in a JPA Query

```
import org.eclipse.persistence.config.HintValues
 import oorg.eclipse.persistence.config.QueryHints;
 query.setHint(QueryHints.REFRESH_CASCADE, CascadePolicy.CascadeAllParts);
```

Example 4–27 shows how to use this hint with the `@QueryHint` annotation.

Example 4–27 Using refresh.cascade in a @QueryHint Annotation

```
import org.eclipse.persistence.config.HintValues;
 import org.eclipse.persistence.config.QueryHints;
 @QueryHint(name=QueryHints.REFRESH_CASCADE, value=CascadePolicy.CascadeAllParts);
```

See Also

For more information, see:

- "EclipseLink" JPA Query Hints
  http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Hints
- "refresh" on page -26

5

eclipselink-orm.xml Schema Reference

You can use EclipseLink's native metadata XML file, `eclipselink-orm.xml`, to override mappings defined in the JPA configuration file (`orm.xml`) and to provide extended ORM features.

> **Note:** Using the `eclipselink-orm.xml` mapping file enables many EclipseLink advanced features, but it may prevent the persistence unit from being portable to other JPA implementations.

The `eclipselink-orm.xml` file defines object-relational mapping metadata for EclipseLink. It has the same basic structure as the `orm.xml` file, which makes it more intuitive, requires minimum configuration, and makes it easy to override.

For more information, see:

- Section 12.2 "XML Overriding Rules" in the JPA Specification
- `http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Configuration/JPA/orm.xml`

The schema for EclipseLink is `eclipselink_orm_X_X.xsd` where X_X is the current EclipseLink version number (such as `2_4` for **2.4**). All EclipseLink schemas are available from `http://wiki.eclipse.org/EclipseLink/XSDs`.

This chapter contains the following sections:

Overriding and Merging

To override the `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` will 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 overridden as determined by the rules on page 5-2.

See the following sections for more information:

Rules for Overriding and Merging

EclipseLink provides specific overriding and merging rules for the following elements defined in the `orm.xml` file:

Persistence Unit Metadata

In `eclipselink-orm.xml`, a `persistence-unit-metadata` element merges or overrides the values of existing `persistence-unit-metadata` specification as defined in Table 5–1.

Table 5–1 Overriding and Merging Persistence Unit Metadata

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 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 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 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 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 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 the `entity-mappings` section. The `eclipselink-orm.xml` entities, embeddables and mapped superclasses are added to the persistence unit as defined in Table 5–2.

Table 5–2 Overriding and Merging Entity Mappings

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 Superclasses

A mapped-superclass can be defined completely, or with specific elements to provide extensions to a mapped-superclass from another mapping file. Table 5–3 lists individual override and merging rules:

Table 5–3 Overriding and Merging Mapped Superclasses

entity-mappings/mapped-superclass	Rule	Description
id-class	Full override	If an id-class exists, then the eclipselink-orm.xml id-class setting overrides the existing setting, or creates a new id-class setting.
exclude-default-listeners	Full override	If an exclude-default-listeners exists, then the eclipselink-orm.xml 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 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 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 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 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 (such as id, embedded-id, basic, version, many-to-one, one-to-many, or one-to-one) exist at the mapping level, then the eclipselink-orm.xml attributes merges or overrides the existing settings, else creates new attributes.

Table 5–3 (Cont.) Overriding and Merging Mapped Superclasses

entity-mappings/mapped-superclass	Rule	Description
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:

Table 5–4 Overriding and Merging Entities

entity-mappings/entity	Rule	Description
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 .
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 5–4 (Cont.) Overriding and Merging Entities

entity-mappings/entity	Rule	Description
`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 any 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. 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.

Table 5–4 (Cont.) Overriding and Merging Entities

entity-mappings/entity	Rule	Description
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	f 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) 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 mapping level 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 wholly or may be defined so as to provide extensions to an embeddable from another mapping file. Therefore, we will allow the merging of that class' metadata. Table 5–4 lists the individual override rules Embeddable classes.

Table 5–5 Overriding and Merging Embeddable Classes

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.

Examples of Overriding and Merging

Example 5–1 Overriding/Merging Example 1

In this example, your EclipseLink project contains:

- `META-INF/orm.xml` – Defines Entity **A** with the mappings **b** and **c**
- `META-INF/eclipselink-orm.xml` – Defines Entity **A** with the mappings **c** and **d**

Results in:

- Entity **A** containing:
  - mapping **b** (from `orm.xml`)
  - mappings **c** and **d** (from `eclipselink-orm.xml`)

Example 5–2 Overriding/Merging Example 2

In this example, your EclipseLink project contains:

- `META-INF/orm.xml` – Defines Entity **A** with 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 **c** and **d**, and Entity **B** with mapping **b** and **c**

Results in:

- Entity A containing:
  - mapping **b** (from `orm.xml`)
  - mappings **c** and **d** (from `eclipselink-orm.xml`)
- Entity B containing:
  - mapping **a** (from `some-other-mapping-file`)
  - mappings **b** and **c** (from `eclipselink-orm.xml`)

Example 5–3 Overriding/Merging Example 3

In this example, your EclipseLink project contains:

- `META-INF/orm.xml` – Defines Entity **A** with mappings **b** and **c**.
- `META-INF/eclipselink-orm.xml` – Defines Entity **A** with mappings **c** and **d**.
- `META-INF/some-other-mapping-file.xml` – Defines Entity **A** with mapping **x**.

Results in:

- Entity **A** containing:
  - mapping **b** (from `orm.xml`)
  - mappings **c** and **d** (from `eclipselink-orm.xml`)
  - mapping **x** (from `some-other-mapping-file.xml`)

Example 5–4 Overriding/Merging Example 4

In this example, your EclipseLink project contains:

- `META-INF/orm.xml` – Defines Entity **A** with mappings **b** and **c**.
- `META-INF/extensions/eclipselink-orm.xml` – Defines defines Entity **A** with mappings **c** and **d**.

  Note: The file is added through a `<mapping-file>` tag in the `persistence.xml` file.

Results in an exception, due to conflicting specifications for mapping **c**.

Example 5–5 Overriding/Merging Example 5

In this example, your EclipseLink project contains:

- `META-INF/orm.xml` – Defines Entity **A** with mappings **b** and **c**
- `META-INF/jpa-mapping-file.xml` – Defines Entity **A** with mappings **a** and **d**
- `META-INF/extensions/eclipse-mapping-file.xml` – Defines defines Entity **A** with mappings **c** and **d**

Results in an exception, due to conflicting specifications for mapping **c** or **d** (which ever is processed first).