DZone Refcardz

CONTENTS INCLUDE:



# EclipseLink JPA

## An Advanced ORM Persistence Framework

*By Gordon Yorke*

## WHAT IS ECLIPSELINK?

Focused on standards, EclipseLink delivers a comprehensive open-source Java persistence solution with implementations of Java Persistence API (JPA), Java Architecture for XML Binding (JAXB), and Service Data Objects (SDO). EclipseLink also offers many advanced features to aid users with their applications.

### EclipseLink JPA

The focus of this Refcard is EclipseLink's object to relational mapping functionality and its extensions of the Java Persistence API specification. This card assumes you are already familiar with the Java Persistence API.

EclipseLink has been JPA 1.0 (JSR 220) certified and is the reference implementation for JPA 2.0 (JSR 317). An offshoot of the code base called TopLink Essentials was the reference implementation for JPA 1.0 (JSR 220).

### How do I get it?

#### Jars and Bundles

EclipseLink is available in two forms: a single eclipselink.jar file that contains all that is needed to use any of the EclipseLink functionality or OSGi bundles for each of the EclipseLink component modules.

Download these from: http://www.eclipse.org/eclipselink/downloads

#### Maven

There are many versions and component module bundles available in the EclipseLink maven repository. Each component module has its own artifactId. For a comprehensive view of what is available, see http://wiki.eclipseLink/Maven. If you just want the eclipselink.jar file, here are example pom.xml entries:

```
<dependencies>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>2.2.0</version>
    <scope>compile</scope>
      ...
  </dependency>
</dependencies>

<repositories>
 <repository>
  <id>EclipseLink Repo</id>
    <url>http://download.eclipse.org/rt/eclipselink/maven.repo</url>
 </repository>
      ...
</repositories>
```

### How to use it

#### Java EE and SE

The eclipselink.jar file or combinations of the OSGi bundles can be treated like normal JAR files and placed in the classpath of your application or server lib.

In a Java EE compliant server, the eclipselink.jar can be bundled in the WAR or EAR file if not already present within the server library.

It is a good practice to place the javax.persistence jar file with the EclipseLink jar.

#### OSGi Environments

To run in a true OSGi environment, the recommended approach is to download the bundles from the Gemini JPA project. These bundles include implementations of the OSGi specification that support dynamic access to the Persistence Providers.

Download the bundles from http://www.eclipse.org/gemini/jpa/. Within the download, there is a GettingStarted.txt file that will help you work through the details of deploying in an OSGi environment.

### Weaving

When deployed within a Java EE server or other SPI supporting servers, EclipseLink has an opportunity to enhance the java classes automatically. This enhancement step, referred to as "weaving", allows EclipseLink to provide many optimizations and improve functionality. For instance, EclipseLink can offer lazy OneToOne and ManyToOne relationships, on-the-fly change tracking and dynamic fetch groups with weaving.

If you are deploying EclipseLink in an environment where weaving is not automatic (as is the case in Java SE), there are two options for enabling weaving.

The first and easiest way to enable weaving is to use the -javaagent: Virtual Machine argument providing the location of the eclipselink.jar. With the java agent specified, EclipseLink will automatically handle weaving of the Entities.

```
java –javaagent:../lib/eclipselink.jar …
```

The second way to enable weaving is to use the "static weaver". The static weaver is a compile time step that is useful in situations where runtime weaving is not available or not desirable.

```
java org.eclipse.persistence.tools.weaving.jpa.StaticWeave -classpath c:\my-pu-jar.jar;<other classpath entries> c:\my-pu-jar.jar c:\my-pu-jar-woven.jar
```

The Static Weave tool can run against directories as well as jars. If the persistence.xml is in a non-standard location, the –persistenceinfo argument can be used to point to the persistence.xml file.

When using static weaving, the Persistence Unit property "eclipselink.weaving" should be set with a value of "static".

Details on other advanced weaving properties can be found here: http://wiki.eclipse.org/Using_EclipseLink_JPA_Extensions_(ELUG)

## Canonical Metamodel

With Java Persistence API 2.0 came the Criteria API for dynamically defining queries. One of the components of the Criteria API is the Canonical Metamodel. The Canonical Metamodel is a collection of generated classes whose static attributes allow a query writer to reference strongly typed Entity metadata for the definition of strongly typed Criteria queries.

```
criteriaQuery.from(Employee.class).join(Employee_.phoneNumbers);
```

Many IDEs, including Eclipse through the Dali project, can generate a Canonical Metamodel for you. However, if this is unavailable, EclipseLink can easily generate these classes during compile time by simply adding the eclipselink-jpa-modelgen_<version>.jar to the javac classpath. For advanced or non-standard usage, the Canonical Metamodel generator has numerous options that are detailed in the EclipseLink user guide.

## Feature Extensions

Many of EclipseLink's extensions have been exposed through annotations and custom xml elements as well as through custom APIs. The classes org.eclipse.persistence.jpa.JpaQuery and org.eclipse.persistence.jpa.JpaEntityManager provide many of the API extensions. org.eclipse.persistence.jpa.JpaHelper has many static methods to help unwrap the Java Persistence API artifacts.

### eclipselink-orm.xml

The eclipselink-orm.xml contains the Eclipselink annotation extensions mentioned in this document in xml format. This file is placed on the application class path at the same locations as the specification defined orm.xml. All Java Persistence API xml and EclipseLink extensions can be defined in this single file or a combination of orm.xml and eclipselink-orm.xml and annotations.

**Hot Tip** All EclipseLink annotations have corresponding elements in the eclipselink-orm.xml file.

# CACHING

Caching provides very efficient data retrieval and can greatly improve your application's performance. Caching is a "first class" feature in EclipseLink and is an integral part of the product. By default, all Entities are cached as fully realized POJOs.

**Hot Tip** When updating a bi-directional relationship (OneToMany/ManyToOne or ManyToMany), ensure both Entities are updated. Setting only the "owning" ManyToOne or ManyToMany mapping will result in Entities missing from the OneToMany collection when retrieved from the cache.

## Configuring the Cache

Using the Java Persistence APIs, you can configure the cache at the Entity level with the @Cacheable annotation and corresponding XML. Using EclipseLink's cache framework, you can finely configure each Entity using the @Cache annotation.

### @Cache Attributes

| | |
|---|---|
| type | cache type. |
| isolation | Degree to which cached instances are shared with other threads. |
| expiry | Invalidates cache entries after a fixed number of milliseconds. |
| expiryTimeOfDay | Invalidates cache entries at specific time of day. |
| alwaysRefresh | Update cache with any data retrieved from the database. Generally used by queries that can not be executed against the cache. |
| refreshOnlyIfNewer | Used with the above but only refresh if the database version is newer than the cached version. |
| disableHits | This will force all queries to execute against the database but return results from the cache. Can be used in combination with "alwaysRefresh". |
| coordinationType | Strategy for communicating instance changes to coordinated nodes. Cache Coordination services must be configured at the EntityManagerFactory level. |

### EclipseLink Cache Types

| | |
|---|---|
| CacheType.FULL | Will cache all Entity instances indefinitely. |
| CacheType.SOFT | Cache uses SoftReference to hold Entities so Cache size is as large as VM allows. |
| CacheType.WEAK | Cache uses WeakReference to hold entities so cache size is dependent on how many Entities are referenced by the application. |
| CacheType.SOFT_WEAK | **Recommended**. A cache of specific size that uses SoftReference to hold a specified number of Entities. Remaining Entities are held by weak references. |
| CacheType.HARD_WEAK | **Recommended**. A cache of specific size that uses "hard" references to hold a specified number of Entities. Remaining Entities are held by weak references. |
| CacheType.CACHE | **Legacy** cache type that is not generally recommended. Can be used by Entities that are not referenced by any other Entity. Cache has an absolute fixed size. |
| CacheType.NONE | **Legacy and Not Recommended**. Can be used by Entities that are not referenced by any other Entities. No caching occurs. |

## What to Cache

Caching can also introduce some challenges to your application. Choosing what and how much to cache is one of the first steps. Take into consideration the volatility of the data, who is changing the data, how distributed the application is, and how sensitive the application is to stale data.

## Stale Data

Stale data occurs when a client has a view of the data that has been changed since it was read. A concurrent application will always have some amount of stale data, regardless of caching, as concurrent clients update the data after other clients have read the data. Generally, an EclipseLink cache does not increase exposure to stale data; however, additional servers, virtual machines, or third-party updates to the data will require a plan. This plan will be a combination of cache size and cacheability of entities combined with appropriate locking, refreshing, and invalidation policies.

### Cache Invalidation Policies

EclipseLink offers automatic refreshing of Entites based on a time of day or age within the cache. @Cache(expiry=<ms>) can be used to specify the age in milliseconds after which Entities of this type will need to be refreshed. @Cache(expiryTimeOfDay=@TimeOfDay(hour, minute, second)) can be used to specify a time of day (for instance, after a third-party nightly batch process has run) after which all Entities within the cache become invalid. Invalid Entities are automatically refreshed by EclipseLink as they are queried.

## Dealing with Multiple Clients

With multiple clients updating and reading data at the same time, it is important to prevent any data corruption when users update stale data or perform updates based on related stale data.

### Optimistic Locking

Versioning each set of updates to an Entity and tracking and comparing those versions is one way to prevent updating stale data. The Java Persistence API provides tracking the version of an Entity through an attribute of the Entity. EclipseLink also provides the functionality to store the version value for the user outside of an Entity class. This can be useful when users do not want to expose versioning within the Entity. It is currently configured through a @DescriptorCustomizer.

The Java Persistence API offers only two supported types of versioning: integer and timestamp mapped to a single attribute. Through the @OptimisticLocking annotion, Eclipselink also supports multi-field locking. With multi-field locking, a set list of fields, changed fields, or all fields can be compared to determine the

version of an Entity. This is useful in legacy systems where a version column has not been made available.

### Refreshing

Once a version conflict has been detected or to ensure a stale data sensitive operation has the latest data, the application can request a refresh of the latest data from the database. The EclipseLink Query Hint "eclipselink.refresh" can be used in combination with queries or EntityManager find and lock operations to cause the refresh to occur more efficiently.

### Cache Coordination

In an environment where multiple instances of the application are deployed, EclipseLink can coordinate the caches of those applications with simple configuration. With cache coordination, active EclipseLink nodes will broadcast either a summary of the changes made to an Entity or just the Entity id, for invalidation, to the other connected caches. With this functionality, the number of Optimistic Lock exceptions experienced can be reduced, which allows an active application to continue to benefit from caches.

Cache coordination is activated through Persistence Unit properties. First, the protocol is set through eclipselink.cache.coordination.protocol:

| | |
|---|---|
| jms | Sends and receives changes through a JMS Topic. |
| jms-publish | Sends changes through JMS, receive through Message Driven Bean. |
| rmi | Uses a fully connected RMI graph to send and receive changes. |
| rmi-iiop | Same as above but uses RMI-IIOP transport protocols. |

Additional configuration is completed based on the protocol selected:

```
<properties>
  …
  <property name="eclipselink.cache.coordination.protocol" value="jms" />
  <property name="eclipselink.cache.coordination.jms.host" value="t3://
localhost:7001/" />
  <property name="eclipselink.cache.coordination.jms.topic" value="jms/
EclipseLinkTopic" />
  <property name="eclipselink.cache.coordination.jms.factory" value="jms/
EclipseLinkTopicConnectionFactory" />
…
</properties>
```

### Distributed Caches

Although there are no provided integrations with distributed caches within the EclipseLink project, the ability to connect to distributed caches is available through org.eclipse.persistence.sessions.interceptors.CacheInterceptor and @CacheInterceptors. Oracle's TopLink Grid product, which integrates with Oracle Coherence, leverages this functionality.

## Not Caching Relationships

An extension to the normal entity cache configuration allows the developer to mark a relationship so that EclipseLink will not cache it. As an extension to the normal Entity cache configuration the @Noncacheable annotation can mark a relationship so that it will not be cached. Each time the Entity is loaded into a Persistence Context, the relationship will be rebuilt. This can be useful if different users have different access rights or if you always want the relationship to be refreshed.

```
@Noncacheable
@OneToOne
protected EntityB entityB;
```

# QUERY EXTENSIONS

EclipseLink has many advanced query features that provide far more flexibility than what is currently in the Java Persistence specification. These features are easily accessible through Query Hints.

## Caching Query Results

If your application repeatedly executes the same queries and these queries are expected to return the same results, EclipseLink can cache the query results. Cached query results eliminate the need to query the database, which greatly improves throughput. As with the Entity cache, the Query cache can be configured for size and to auto-expire.

The Query Cache can be configured through Query Hints within the Java Persistence API.

| | |
|---|---|
| eclipselink.query-results-cache | Activates the query cache for this query. |
| eclipselink.query-results-cache.type | Has same cache types as the Entity cache but the default for the Query cache is CacheType.cache. |
| eclipselink.query-results-cache.size | Sets a cache size if appropriate. |
| eclipselink.query-results-cache.expiry | Expires cache entries after a fixed number of milliseconds. |
| eclipselink.query-results-cache.expiry-time-of-day | Expires cache entries at specific time of day. |

## Bulk Reading

If your application needs to read in multiple related Entities it can be far more efficient to read these Entities in a few queries rather than issue multiple separate queries for each relationship and query result. Java Persistence API allows for limited bulk reading using the JOIN FETCH construct in JPQL. However, EclipseLink offers both nested joins and Batch reading.

### Join Fetch

Using the Query Hint eclipselink.join-fetch and a simple dot notation, an application can request multiple levels of Entities to be join fetched.

```
Query query = entityManger.createQuery(
                                "SELECT e FROM Employee e WHERE …");
query.setHint("eclipselink.join-fetch", "e.department.address");
```

This example will select the Employee, the employee's Department, and the department's Address in a single query.

## Batch Reading

EclipseLink has an additional bulk reading feature called Batch Reading. Using Join Fetch can often require significant resources as a great deal of duplicate data is returned (for instance, when joining in a OneToMany relationship). Querying the related data using Batch Reading can be more efficient. Batch Reading will query for the root Entity and then use a subsequent query to load the related data. Although there is an additional query, it reduces the number of joins on the database and the amount of data returned.

```
Query query = entityManger.createQuery(
                                "SELECT e FROM Employee e WHERE …");
query.setHint("eclipselink.batch", "e.phoneNumbers");
```

**Hot Tip** Batch Reading can be used in combination with Join Fetch for optimized bulk reading.

## Stored Procedures

Executing a Stored Procedure through EclipseLink's Java Persistence API extensions is as easy as defining a Named Stored Procedure query.

```
@NamedStoredProcedureQuery(
    name="SProcAddress",
    resultClass=models.jpa.advanced.Address.class,
    procedureName="SProc_Read_Address",
    parameters={
        @StoredProcedureParameter(
            direction=IN_OUT,
            name="address_id_v",
            queryParameter="ADDRESS_ID",
            type=Integer.class),
        @StoredProcedureParameter(
            direction=OUT,
            name="street_v",
            queryParameter="STREET",
            type=String.class)
    }
)
```

This query is then called using the Java Persistence APIs.

```
Query aQuery = em.createNamedQuery("SProcAddress")
```

A call to getResultList() will return a collection of Address entities through the Stored Procedure's result set. If no resultset is returned,

EclipseLink can build Entity results from the output parameters if the resultClass or resultSetMapping is set.

The Direction.OUT_CURSOR is used for stored procedure parameters that are returning resultsets through "ref cursors".

If both result sets and output parameters are returned by the stored procedure, then a SessionEventListener that responds to outputParametersDetected should be registered with the EntityManagerFactory and the output parameters will be returned through this event.

## Fetch Groups

Lazy attributes can make queries more efficient by delaying the loading of data. EclipseLink offers a feature called Fetch Groups that allows you to define multiple lazy attribute configurations and apply those on a per-query basis. Attributes are fully lazy accessible, and accessing an unfetched attribute will cause the Entity attributes to be loaded as per the static lazy configuration from the Entity metadata.

### Configuration

Fetch Groups can be created statically using @FetchGroup.

```
@FetchGroups({
    @FetchGroup(
        name="FirstLast",
        attributes={
            @FetchAttribute(name="first"),
            @FetchAttribute(name="last")
        }
    ),
    @FetchGroup(
        name="LastName",
        attributes={@FetchAttribute(name="lastName")}
    )
})
```

They are then applied to the query through the use of Query Hint.

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

Fetch Groups can also be created and used at runtime…

```
FetchGroup fetchGroup = new FetchGroup();  fetchGroup.addAttribute("lastName");
query.setHint(QueryHints.FETCH_GROUP, fetchGroup);
```

Other Fetch Group query hints are available as well to provide fine-grained configuration. Details can be found in the QueryHints javaDocs.

**Hot Tip** Have multiple Persistence Contexts in the same transaction? If any are performing transactional writes, ensure you disable an EclipseLink optimization with the Entity Manager property eclipselink.transaction.join-existing so all queries will access transactional data.

## Cursors

When reading a large dataset, the application may wish to stream the results from a query. This is useful if the dataset is quite large. Using the query hint "eclipselink.cursor" set to "true", EclipseLink will return a org.eclipse.persistence.queries.CursoredStream, which will allow the results to be retrieved iteratively from the database.

**Hot Tip** Need to load a large dataset and want to bypass the Persistence Context to save heap space? Use the Query Hint "eclipselink.read-only" and the results of the Query will not be managed.

## Query Redirectors

Should an application have the need to dynamically alter a query or execute a query against another resource, Query Redirectors can be used. A redirector implements a simple interface org.eclipse.persistence.queries.QueryRedirector, and the developer of the Redirector is free to do most anything. Any cache maintenance will have to occur within the Redirector if the query is executed against another resource.

### Configuration

Query Redirectors can be set through the Query Hint eclipselink.query.redirector or set as default Redirectors on an Entity.

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

Default Redirectors will be called every time a query is executed against this Entity type. Default Redirectors can be configured per Entity by Query type as well.

# MAPPINGS

## Converters

Sometimes the data in the database column does not match the type of the corresponding Entity attribute. EclipseLink can handle simple conversion for @Basic mappings like Number to String but what if the conversion is more complicated? This is where converters can be useful. With a converter, custom bi-directional conversion can occur between the database and the Entity attributes. Converters are supported on @Basic mappings and primitive @ElementCollections.

EclipseLink offers some predefined converters that support commonly occurring conversions:

| | |
|---|---|
| @ObjectTypeConverter | Maps column values to attribute values (ie, "t" -> True). |
| @TypeConverter | Converts database type to Java type. Useful for ElementCollections. |
| @StructConverter | Converts from Database Struct type. JGeometry has built in support. |

For truly custom conversions, the @Converter that references an implementation of the org.eclipse.persistence.mappings.converters.Converter interface can be specified.

Converters can be set directly on the mapping through the corresponding annotation, or a Converter can be defined at the Entity level and referenced by name through @Convert. If the ElementCollection is a Map type, then a Converter can be applied to the Map's Key through @MapKeyConvert.

```
@TypeConverter(
    name="Long2String",
    dataType=String.class,
    objectType=Long.class
)
@ObjectTypeConverter(
       name="CreditLine",
       conversionValues={
           @ConversionValue(dataValue="RB",
                            objectValue=ROYAL_BANK),
           @ConversionValue(dataValue="CIB",
                            objectValue=CANADIAN_IMPERIAL),
           @ConversionValue(dataValue="SB",
                            objectValue=SCOTTSBANK),
           @ConversionValue(dataValue="HD",
                            objectValue=HALIFAX_DOWNTOWN)
       }
)
@Entity
public class Entity …
    @ElementCollection
    @MapKeyColumn(name="BANK")
    @Column(name="ACCOUNT")
    @Convert("Long2String")
    @MapKeyConvert("CreditLine")
    public Map<String, Long> getCreditLines() {
        return creditLines;
    }
```

## Interface Attribute Types

If an application has single-valued Entity attributes that are interface types, EclipseLink's VariableOneToOne mapping can be used to map these attributes as a polymorphic relationship. This mapping type uses a discriminator column similar to Java Persistence API inheritance support; but, in this case, the discriminator is found on the source table.

```
    @VariableOneToOne(
            targetInterface=Distributor.class,
            cascade=PERSIST,
            fetch=LAZY,
            discriminatorColumn=@DiscriminatorColumn(
                                        name="DISTRIBUTOR_TYPE",
                                        discriminatorType=INTEGER),
            discriminatorClasses={
                @DiscriminatorClass(discriminator="1",
                                    value=MegaBrands.class),
                @DiscriminatorClass(discriminator="2",
                                    value=Namco.class)
            }
    )
    public Distributor getDistributor() {
        return distributor;
    }
```

To execute queries with path expressions that include an interface mapping, a Query Key must be defined on the Interface descriptor. Currently, this must done using a SessionCustomizer. Details on Query Keys can be found here: http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Query_Keys

## Additional Filtering Criteria

Need to support multi-tenancy, temporal filtering or any number of other use cases where an application has a query restriction that needs to be applied to the restrictions on all queries? @AdditionalCriteria can be applied to an Entity class to specify that additional filtering occurs for any queries on that type. When specifying the path in an AdditionalCriteria, the identification variable "this" is used to identify the root of the query.

```
@AdditionalCriteria("this.role = :accessRole");
@Entity
public class EntityA {…}
```

Additional Criteria supports parameters, and parameter values can be set at the EntityManagerFactory or EntityManager levels.

```
entityManager.setProperty("accessRole", "bravo");
```

**Hot Tip** Combine Additional Criteria with @Noncacheable relationships to provide EntityManager scoped user data.

## Return On Insert/Update

If the database has triggers that return data to the calling client, the @ReturnInsert / @ReturnUpdate annotations can be applied to the corresponding @Basic mappings to have EclipseLink apply the return value to the Entity.

The result from the database will be parsed for the mappings field and the value placed in the attribute.

```
@ReturnInsert(returnOnly=true)
@Basic
protected long id;
```

# CONNECTIONS

## Partitioning

If an application needs access to multiple databases (perhaps for load balancing or data affinity), EclipseLink has functionality called Partitioning that will allow many schemes for accessing multiple databases.

With partitioning, users can "stripe" data across multiple databases or schemas by Entity field value or by Entity type.

```
@RangePartitioning, @HashPartitioning, @PinnedPartitioning, @ValuePartitiong
```

Other Partitioning Policies are available for load balancing or replication across multiple databases or schemas.

```
@ReplicationPartititoning, @RoundRobinPartitioning, @UnionPartitioning
```

Entirely custom policy implementations are also an option. If users need certain processing that can not be found in the pre-existing partitioning implementations the PartitioningPolicy can be extended and a custom Policy specified using @Partitioning.

An example of an Entity's partitioning configuration follows:

```
@Entity
@Table(name = "PART_DEPT")
@HashPartitioning(
    name="HashPartitioningByID",
    partitionColumn=@Column(name="ID"),
    unionUnpartitionableQueries=true,
    connectionPools={"node2", "node3"})
@UnionPartitioning(
        name="UnionPartitioningAllNodes",
        replicateWrites=true)
@Partitioned("HashPartitioningByID")
public class Department implements Serializable {
```

## VPD, Proxy Authentication

If an application needs to access a database when proxy authentication or row level security is required, EclipseLink has easy-to-use configuration that will allow "authentication" of the connection.

For Proxy Authentication, provide the necessary Persistence Unit properties to the EntityManagerFactory or EntityManager and EclipseLink will ensure the connection acquired from the datasource will be "authenticated".

```
Map emProperties = new HashMap();
emProperties.put("eclipselink.oracle.proxy-type",
                 OracleConnection.PROXYTYPE_USER_NAME);
emProperties.put(OracleConnection.PROXY_USER_NAME, "john");
EntityManager em = emf.createEntityManager(emProperties);
```

Or in the case of injection:

```
entityManager.setProperty("eclipselink.oracle.proxy-type",
                          OracleConnection.PROXYTYPE_USER_NAME);
entityManager.setProperty(OracleConnection.PROXY_USER_NAME, "john");
```

For Virtual Private Database (VPD) or manual "authentication", a Session Event exists "postAcquireExclusiveConnection" that will be called whenever EclipseLink acquires a connection for "authentication". This event can be used to provide any required details to the connection like a ClientIdentifier.

An additional property that specifies when EclipseLink must use the "authenticated" connection is required for both VPD type and Proxy authentication. "eclipselink.jdbc.exclusive-connection.mode" should be set to "Isolated" if all Entities corresponding to secured tables have been set to @Cacheable(false) or to "Always" otherwise .

## Connection Pooling

When deploying an EclipseLink application in an environment without managed datasources, EclipseLink offers built-in connection management. Configuring connection pooling can be as simple as setting the Persistence Unit Properties using "eclipselink.connection-pool. " during creation of the Entity Manager Factory. There are many connection pool properties combinations that can be found in the JavaDocs.

# OTHER POPULAR EXTENSIONS

## Persistence Context Memory Management

If a Persistence Context tends to have a long lifecycle within an application, managing the size of the Persistence Context can be difficult. EclipseLink can allow an application to leverage the Java garbage collector and remove Entities from the Persistence Context that are no longer in use by the application. This is done through the EntityManager property "eclipselink.persistence-context.reference-mode". Possible values are:

- HARD - the default and ensures no Entities will be removed from the Persistence Context automatically.
- WEAK – in combination with weaving, ensures any Entities with changes will not be removed from Persistence Context; but other unreferenced Entities may be removed.
- FORCED_WEAK – any unreferenced Entities may be removed from the Persistence Context by the garbage collector. Changes in these Entities will be lost if not already flushed.

## Batch Writing

EclipseLink supports writing to the database using batched writes. This is configured through the Persistence Unit property "eclipselink.jdbc.batch-writing" and should be used in combination with parameter binding, which is active by default. Possible values for the property are:

- JDBC – this uses standard JDBC batch writing and should be used in most cases.
- Oracle-JDBC – use native Oracle database batch writing.

## Customizers

EclipseLink has a great depth to its functionality. Although the most popular features are available, not all of the EclipseLink functionality is exposed through JPA extensions. "Customizers" are a simple mechanism that exposes internal EclipseLink customization to Java Persistence API applications.

Both EntityManagerFactory level customizers, called Session Customizers, and Entity level mapping customizers, called Descriptor Customizers, are available.

To use a Session Customizer, the Persistence Unit property "eclipselink.session.customizer" is used to specify an implementor of the interface org.eclipse.persistence.config.SessionCustomizer.

To use a Descriptor Customizer, a Persistence Unit property "eclipselink.descriptor.customizer.<entity name>" or @Customizer is used to specify a class that implements the interface org.eclipse.persistence.config.DescriptorCustomizer.

## Event Listeners

EclipseLink has far more events available than those specified by Java Persistence API. Events are available on query execution, Entity Manager Factory creation, stored procedure output parameter results, connection acquisition, precise Entity lifecycle events, and at many other points during a running application.

### Entity Listeners

The easiest listeners to use in Java Persistence are the Entity Listeners. These listeners (such as aboutToUpdate, postUpdate, postDelete, etc ) provide events at points during an Entity's lifecycle. Events like these can be useful to implement auditing or client notification. To receive the event notification, the interface org.eclipse.persistence.descriptors.DescriptorEventListener is implemented or org.eclipse.persistence.descriptors.DescriptorEventAdaptor is extended for the desired events. Then using @EntityListeners, these listener classes are provided to EclipseLink.

### Entity Manager Factory Listeners

The Entity Manager Factory Listeners are referred to as SessionEventListeners in EclipseLink. These listeners are notified for more general process notifications, like connection acquisition that is used for VPD; transaction events like preRollbackTransaction; EntityManager process events like postCalculateChanges that are fired after flush but before writes have begun; query execution events like outputParametersDetected for Stored Procedure execution; and many more events.

To receive notification of these events, a custom class is created that implements org.eclipse.persistence.sessions.SessionEventListener or extends org.eclipse.persistence.sessions.SessionEventAdaptor for the desired events. Then the listener is set on the Entity Manager Factory through the Persistence Unit property "eclipselink.session-event-listener".

## RESOURCES

## Community

EclipseLink is a very powerful product with many customization and extension points, all of which could not possibly be covered in a Refcard. For more information or help using a feature, EclipseLink has an active and helpful community. Help can always be found using eclipselink-users@eclipse.org or the forums http://www.eclipse.org/forums/index.php?t=thread&frm_id=111. A comprehensive User Guide can be found here: http://wiki.eclipse.org/EclipseLink/UserGuide

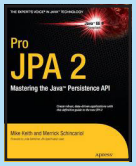
### Enhancements and Feedback

Because EclipseLink is an open-source project, new feature development, designs, and bug reports are available for review and feedback. eclipselink-dev@eclipse.org is an active mailing list, and all feature development is tracked using wiki http://wiki.eclipse.org/EclipseLink/Development and the bug database https://bugs.eclipse.org/bugs

## ABOUT THE AUTHOR

Gordon Yorke is an EclipseLink Architecture Council member and committer, JSR 317 & JSR 338(JPA 2.0 & 2.1) Expert Group member and a long time developer of EclipseLink and its past permutations. With over 11 years of ORM framework experience Gordon brings a wealth of knowledge on object-relational persistence, data-access and caching.

## RECOMMENDED BOOK




*Pro JPA 2* is a detailed learning and use reference, written by EJB co-spec lead and JPA contributor Mike Keith and his colleague.



DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.



DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

**Refcardz Feedback Welcome**
refcardz@dzone.com

**Sponsorship Opportunities**
sales@dzone.com

ISBN-13: 978-1-936502-44-8
ISBN-10: 1-936502-44-5
50795
9 781936 502448
$7.95

Version 1.0