Jakarta Persistence Guide (v6.0)

Table of Contents

We saw in Jakarta Mapping Guide how to map classes for persistence with the Jakarta Persistence API. In this guide we will describe the Jakarta Persistence API itself, showing how to persist, update and delete objects from persistence.

You should familiarise yourself with the Jakarta 3.0 Javadocs.

EntityManagerFactory

Any Jakarta Persistence-enabled application will require at least one *EntityManagerFactory* (EMF). Typically applications create one per datastore being utilised. An *EntityManagerFactory* provides access to *EntityManager(s)* which allow objects to be persisted, and retrieved. The *EntityManagerFactory* can be configured to provide particular behaviour.

An *EntityManagerFactory* is designed to be thread-safe. An *EntityManager* is not.

An *EntityManagerFactory* is expensive to create so you should create one per datastore for your application and retain it for as long as it is needed.

Always close your *EntityManagerFactory* / *EntityManager* objects after you have finished with them.

Create an EMF in JavaSE

The simplest way of creating an *EntityManagerFactory* Javadoc in a JavaSE environment is as follows

```
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

...

EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPU");
```

Here we provide the name of the persistence-unit which defines the datastore, properties, classes, meta-data etc to be used. An alternative is to specify the properties to use along with the *persistence-unit* name; in that case the passed properties will override any that are specified for the persistence unit itself.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPU",
overridingProps);
```

Create an EMF in JavaEE

If you want an **application-managed** EMF then you create it by injection like this, providing the name of the required persistence-unit

```
@PersistenceUnit(unitName="myPU")
EntityManagerFactory emf;
```

If you want a **container-managed** EM then you create it by injection like this, providing the name

of the required persistence-unit

```
@PersistenceContext(unitName="myPU")
EntityManager em;
```

Please refer to the docs for your JavaEE server for more details.

Persistence Unit

As shown above, we create an EMF for a *persistence-unit*. A *persistence-unit* is simply a way of having independent groupings of entities, mapping info and/or jars that will be managed together. The *persistence-unit* is named, and the name is used for identifying it (as used above in creating the EMF). This name can also then be used when defining what classes are to be enhanced, for example.

To define a *persistence-unit* you first need to add a file `persistence.xml` to the `META-INF/` directory of your application jar. This file will be used to define your *persistence-unit(s)*. Let's show an example

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
        https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd" version="3.0">

    <!-- Online Store -->
    <persistence-unit name="OnlineStore">
        <provider>org.datanucleus.api.jakarta.PersistenceProviderImpl</provider>
        <class>mydomain.samples.metadata.store.Product</class>
        <class>mydomain.samples.metadata.store.Book</class>
        <class>mydomain.samples.metadata.store.CompactDisc</class>
        <class>mydomain.samples.metadata.store.Customer</class>
        <class>mydomain.samples.metadata.store.Supplier</class>
        <exclude-unlisted-classes/>
        <properties>
            <property name="jakarta.persistence.jdbc.url" value=
"jdbc:h2:datanucleus"/>
            <property name="jakarta.persistence.jdbc.user" value="sa"/>
            <property name="jakarta.persistence.jdbc.password" value=""/>
        </properties>
    </persistence-unit>

    <!-- Accounting -->
    <persistence-unit name="Accounting">
        <provider>org.datanucleus.api.jakarta.PersistenceProviderImpl</provider>
        <mapping-file>com/datanucleus/samples/metadata/accounts/orm.xml</mapping-file>
        <properties>
            <property name="jakarta.persistence.jdbc.url" value=
"jdbc:h2:datanucleus"/>
            <property name="jakarta.persistence.jdbc.user" value="sa"/>
            <property name="jakarta.persistence.jdbc.password" value=""/>
        </properties>
    </persistence-unit>

</persistence>
```

In this example we have defined 2 *persistence-unit(s)*. The first has the name "OnlineStore" and contains 5 classes (annotated). The second has the name "Accounting" and contains a metadata file called `orm.xml` in a particular package (which will define the classes being part of that unit). This means that once we have defined this we can reference these *persistence-unit(s)* in our persistence operations. You can find the XSD for `persistence.xml` here.

There are several sub-elements of this `persistence.xml` file worth describing

- **provider** - the Jakarta persistence provider to be used. The Jakarta persistence "provider" for DataNucleus is **org.datanucleus.api.jakarta.PersistenceProviderImpl**
- **jta-data-source** - JNDI name for JTA connections (make sure you set *transaction-type* as **JTA** on the persistence-unit for this) **This is only for RDBMS.**

- **non-jta-data-source** - JNDI name for non-JTA connections. Note that if using a JTA datasource as the primary connection, you ought to provide a *non-jta-data-source* also since any schema generation and/or sequence handling will need to use that **This is only for RDBMS.**
- **shared-cache-mode** - Defines the way the L2 cache will operate. ALL means all entities cached. NONE means no entities will be cached. ENABLE_SELECTIVE means only cache the entities that are specified. DISABLE_SELECTIVE means cache all unless specified. UNSPECIFIED leaves it to DataNucleus.
- **validation-mode** - Defines the validation mode for Bean Validation. AUTO, CALLBACK or NONE.
- **jar-file** - name of a JAR file to scan for annotated classes to include in this persistence-unit.
- **mapping-file** - name of an XML "mapping" file containing persistence information to be included in this persistence-unit. NOTE that the Jakarta Persistence spec defines a default file called `META-INF/orm.xml` that does not need to be specified.
- **class** - name of an annotated class to include in this persistence-unit
- **properties** - properties defining the persistence factory to be used. Please refer to EMF Properties for details

Metadata loading using persistence unit

When you specify an EMF using a `persistence.xml` it will load the metadata for all classes that are specified directly in the persistence unit. If you don't have the *exclude-unlisted-classes* set to true then it will also do a CLASSPATH scan to try to find any other **annotated** classes that are part of that persistence unit. To set the CLASSPATH scanner to a custom version use the persistence property **datanucleus.metadata.scanner** and set it to the classname of the scanner class.

Specifying the datastore properties

With a persistence-unit you have 2 ways of specifying the datastore to use

- **Specify the connection URL/userName/password(/driver)** and it will internally create a DataSource for this URL (or equivalent for non-RDBMS). This is achieved by specifying **jakarta.persistence.jdbc.url**, **jakarta.persistence.jdbc.user**, **jakarta.persistence.jdbc.password**, **jakarta.persistence.jdbc.driver** properties. This optionally includes connection pooling dependent on datastore.
- **Specify the JNDI name of the connectionFactory** (only for RDBMS). This is achieved by specifying **jakarta.persistence.jtaDataSource**, and **jakarta.persistence.nonJtaDataSource** (for secondary operations) or by specifying the element(s) *jta-data-source*/*non-jta-data-source*

The connection "url" value for the different supported datastores is defined in the Datastore Guide

Restricting to specific classes

If you want to just have specific classes in the *persistence-unit* you can specify them using the **class** element, and then add **exclude-unlisted-classes**, like this

```
<persistence-unit name="Store">
    <provider>org.datanucleus.api.jakarta.PersistenceProviderImpl</provider>
    <class>mydomain.samples.metadata.store.Product</class>
    <class>mydomain.samples.metadata.store.Book</class>
    <class>mydomain.samples.metadata.store.CompactDisc</class>
    <exclude-unlisted-classes/>
    ...
</persistence-unit>
```

If you don't include the **exclude-unlisted-classes** then DataNucleus will search for annotated classes starting at the *root* of the *persistence-unit* (the root directory in the CLASSPATH that contains the `META-INF/persistence.xml` file).

Dynamically generated Persistence-Unit

DataNucleus allows an extension to the Jakarta Persistence API to dynamically create persistence-units at runtime. Use the following code sample as a guide. Obviously any *entity classes* defined in the persistence-unit need to have been enhanced.

```
import org.datanucleus.metadata.PersistenceUnitMetaData;
import org.datanucleus.api.jakarta.JakartaEntityManagerFactory;

PersistenceUnitMetaData pumd = new PersistenceUnitMetaData("dynamic-unit",
"RESOURCE_LOCAL", null);
pumd.addClassName("mydomain.test.A");
pumd.setExcludeUnlistedClasses();
pumd.addProperty("jakarta.persistence.jdbc.url", "jdbc:h2:mem:nucleus");
pumd.addProperty("jakarta.persistence.jdbc.user", "sa");
pumd.addProperty("jakarta.persistence.jdbc.password", "");
pumd.addProperty("datanucleus.schema.autoCreateAll", "true");

EntityManagerFactory emf = new JakartaEntityManagerFactory(pumd, null);
```

It should be noted that if you call *pumd.toString();* then this returns the text that would have been found in a `persistence.xml` file.

EntityManagerFactory Properties

An EntityManagerFactory is very configurable, and DataNucleus provides many properties to tailor its behaviour to your persistence needs.

Standard Jakarta Properties

Parameter	Description + Values
jakarta.persistence.provider	Class name of the provider to use. DataNucleus has a provider name of **org.datanucleus.api.jakarta.PersistenceProviderImpl**. If you only have 1 persistence provider in the CLASSPATH then this doesn't need specifying.
jakarta.persistence.transactionType	Type of transactions to use. In Java SE the default is RESOURCE_LOCAL. In Java EE the default is JTA. Note that if using a JTA datasource as the primary connection, you ought to provide a *non-jta-data-source* also since any schema generation and/or sequence handling will need to use that. *{RESOURCE_LOCAL, JTA}*
jakarta.persistence.jtaDataSource	JNDI name of a (transactional) JTA data source. Note that if using a JTA datasource as the primary connection, you ought to provide a *non-jta-data-source* also since any schema generation and/or sequence handling will need to use that.
jakarta.persistence.nonJtaDataSource	JNDI name of a (non-transactional) data source. This is used for schema/value generation operations.
jakarta.persistence.jdbc.url	URL specifying the datastore to use for persistence. Note that this will define the **type of datastore** as well as the datastore itself. Please refer to the Datastore Guide for the URL appropriate for the type of datastore you're using.
jakarta.persistence.jdbc.user	Username to use for connecting to the DB
jakarta.persistence.jdbc.password	Password to use for connecting to the DB
jakarta.persistence.jdbc.driver	The name of the (JDBC) driver to use for the DB (for RDBMS only, and not needed for JDBC 4+ drivers). Note that some 3rd party connection pools do require the driver class name still. For LDAP this would be the initial context factory.
jakarta.persistence.query.timeout	Timeout for queries (global)
jakarta.persistence.sharedCache.mode	The mode of operation of the L2 cache, deciding which entities are cached. The default (UNSPECIFIED) is the same as DISABLE_SELECTIVE. See also Cache docs for Jakarta *{NONE, ALL, ENABLE_SELECTIVE, DISABLE_SELECTIVE, **UNSPECIFIED**}*
jakarta.persistence.validation.mode	Determines whether the automatic lifecycle event validation is in effect. *{**auto**, callback, none}*
jakarta.persistence.validation.group.pre-persist	The classes to validation on pre-persist callback
jakarta.persistence.validation.group.pre-update	The classes to validation on pre-update callback

Parameter	Description + Values
jakarta.persistence.validation.group.pre-remove	The classes to validation on pre-remove callback
jakarta.persistence.validation.factory	The validation factory to use in validation
jakarta.persistence.bean.manager	CDI BeanManager, to enable CDI injection into `AttributeConverter` and event listener objects.
jakarta.persistence.schema-generation.database.action	Whether to perform any schema generation to the database at startup. Will process the schema for all classes that have metadata loaded at startup (i.e the classes specified in a persistence-unit). *{create, drop, drop-and-create, **none**}*
jakarta.persistence.schema-generation.scripts.action	Whether to perform any schema generation into scripts at startup. Will process the schema for all classes that have metadata loaded at startup (i.e the classes specified in a persistence-unit). *{create, drop, drop-and-create, **none**}*
jakarta.persistence.schema-generation.create-source	Specifies the order for create operations. If a script is provided then defaults to "script", otherwise defaults to "metadata". *{script, metadata, script-then-metadata, metadata-then-script}*
jakarta.persistence.schema-generation.scripts.create-target	Name of the script file to write to if doing a "create" with the target as "scripts" *{**datanucleus-schema-create.ddl**, {filename}}*
jakarta.persistence.schema-generation.create-script-source	Name of a script file to run to create tables. Can be absolute filename, or URL string *{filename}*
jakarta.persistence.schema-generation.drop-source	Specifies the order for drop operations. If a script is provided then defaults to "script", otherwise defaults to "metadata". *{script, metadata, script-then-metadata, metadata-then-script}*
jakarta.persistence.schema-generation.scripts.drop-target	Name of the script file to write to if doing a "drop" with the target as "scripts" *{**datanucleus-schema-drop.ddl**, {filename}}*
jakarta.persistence.schema-generation.drop-script-source	Name of a script file to run to drop tables. Can be absolute filename, or URL string *{filename}*
jakarta.persistence.sql-load-script-source	Name of a script file to run to load data into the schema. Can be absolute filename, or URL string *{filename}*

DataNucleus Datastore Properties

DataNucleus provides the following properties for configuring the datastore connection used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.ConnectionURL	Refer to *jakarta.persistence.jdbc.url*.
datanucleus.ConnectionUserName	Refer to *jakarta.persistence.jdbc.user*.
datanucleus.ConnectionPassword	Refer to *jakarta.persistence.jdbc.password*.
datanucleus.ConnectionDriverName	Refer to *jakarta.persistence.jdbc.driver*.
datanucleus.ConnectionFactory	Instance of a connection factory for **transactional** connections. This is an alternative to **datanucleus.ConnectionURL**. **Only for RDBMS**, and it must be an instance of javax.sql.DataSource. **Note that you will also need to define a separate ConnectionFactory2 for schema/sequence operations where those are required**. See Data Sources
datanucleus.ConnectionFactory2	Instance of a connection factory for **nontransactional** connections. This is an alternative to **datanucleus.ConnectionURL**. **Only for RDBMS**, and it must be an instance of javax.sql.DataSource. **Note that you if using ConnectionFactory then you need to define this as a separate factory for schema/sequence operations**. See Data Sources.
datanucleus.ConnectionFactoryName	The JNDI name for a connection factory for **transactional** connections. **Only for RDBMS**, and it must be a JNDI name that points to a javax.sql.DataSource object. See Data Sources.
datanucleus.ConnectionFactory2Name	The JNDI name for a connection factory for **nontransactional** connections. **Only for RDBMS**, and it must be a JNDI name that points to a javax.sql.DataSource object. See Data Sources.
datanucleus.ConnectionPasswordDecrypter	Name of a class that implements *org.datanucleus.store.ConnectionEncryptionProvider* and should only be specified if the password is encrypted in the persistence properties
datanucleus.connectionPoolingType	This property allows you to utilise a 3rd party software package for enabling connection pooling. When using RDBMS you can select from DBCP2, C3P0, HikariCP, BoneCP, etc. You must have the 3rd party jars in the CLASSPATH to use these options. Please refer to the Connection Pooling guide for details. {None, **dbcp2-builtin**, DBCP2, C3P0, BoneCP, HikariCP, Tomcat, {others}}
datanucleus.connectionPoolingType.nontx	This property allows you to utilise a 3rd party software package for enabling connection pooling **for nontransactional connections** using a DataNucleus plugin. If you don't specify this value but do define the above value then that is taken by default. Refer to the above property for more details. {None, **dbcp2-builtin**, DBCP2, C3P0, BoneCP, HikariCP, Tomcat, {others}}

Parameter	Description + Values
datanucleus.connection.nontx.releaseAfterUse	Applies only to non-transactional connections and refers to whether to re-use (pool) the connection internally for later use. The default behaviour is to close any such non-transactional connection after use. If doing significant non-transactional processing in your application then this may provide performance benefits, but be careful about the number of connections being held open (if one is held open per EM). {**true**, false}
datanucleus.connection.singleConnectionPerExecutionContext	With an ExecutionContext (EM) we normally allocate one connection for a transaction and close it after the transaction, then a different connection for nontransactional ops. This flag acts as a hint to the store plugin to obtain and retain a single connection throughout the lifetime of the EM. {true, **false**}
datanucleus.connection.resourceType	Resource Type for primary connection {RESOURCE_LOCAL, JTA}
datanucleus.connection.resourceType2	Resource Type for secondary connection {RESOURCE_LOCAL, JTA}

DataNucleus Persistence Properties

DataNucleus provides the following properties for configuring general persistence handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.IgnoreCache	Whether to ignore the cache for queries. If the user sets this to *true* then the query will evaluate in the datastore, but the instances returned will be formed from the datastore; this means that if an instance has been modified and its datastore values match the query then the instance returned will **not** be the currently cached (updated) instance, instead an instance formed using the datastore values. {true, **false**}
datanucleus.Multithreaded	Whether to try run the EntityManager as multithreaded. **Note that this is only a hint to try to allow thread-safe operations on the EM. Users are always advised to run an EM as single threaded, since some operations are not currently locked and so could cause issues multi-threaded.** {true, **false**}
datanucleus.Optimistic	Whether to use optimistic locking. {**true**, false}
datanucleus.RetainValues	Whether to suppress the clearing of values from persistent instances on transaction completion. {**true**, false}
datanucleus.RestoreValues	Whether persistent object have transactional field values restored when transaction rollback occurs. {true, **false**}

Parameter	Description + Values
datanucleus.mapping.Catalog	Name of the catalog to use by default for all classes persisted using this EMF. This can be overridden in the MetaData where required, and is optional. DataNucleus will prefix all table names with this catalog name if the RDBMS supports specification of catalog names in DDL. **RDBMS only**
datanucleus.mapping.Schema	Name of the schema to use by default for all classes persisted using this EMF. This can be overridden in the MetaData where required, and is optional. DataNucleus will prefix all table names with this schema name if the RDBMS supports specification of schema names in DDL. **RDBMS only**
datanucleus.tenantId	String id to use as a discriminator on all persistable class tables to restrict data for the tenant using this application instance (aka multi-tenancy via discriminator). **RDBMS, MongoDB, HBase, Neo4j, Cassandra only**
datanucleus.tenantProvider	Instance of a class that implements *org.datanucleus.store.schema.MultiTenancyProvider* which will return the tenant name to use for each call. **RDBMS, MongoDB, HBase, Neo4j, Cassandra only**
datanucleus.CurrentUser	String defining the current user for the persistence process. Used by auditing. *RDBMS datastores only*
datanucleus.CurrentUserProvider	Instance of a class that implements *org.datanucleus.store.schema.CurrentUserProvider* which will return the current user to use for each call. Used by auditing. *RDBMS datastores only*
datanucleus.DetachAllOnCommit	Allows the user to select that when a transaction is committed all objects enlisted in that transaction will be automatically detached. {true, **false**}
datanucleus.detachAllOnRollback	Allows the user to select that when a transaction is rolled back all objects enlisted in that transaction will be automatically detached. {true, **false**}
datanucleus.CopyOnAttach	Whether, when attaching a detached object, we create an attached copy or simply migrate the detached object to attached state {**true**, false}
datanucleus.allowAttachOfTransient	When you call EM.merge with a transient object (with PK fields set), if you enable this feature then it will first check for existence of an object in the datastore with the same identity and, if present, will merge into that object (rather than just trying to persist a new object). {**true**, false}
datanucleus.attachSameDatastore	When attaching an object DataNucleus by default assumes that you're attaching to the same datastore as you detached from. DataNucleus does though allow you to attach to a different datastore (for things like replication). Set this to *false* if you want to attach to a different datastore to what you detached from. This property is also useful if you are attaching and want it to check for existence of the object in the datastore before attaching, and create it if not present (*true* assumes that the object exists). {**true**, false}

Parameter	Description + Values
datanucleus.detachAsWrapped	When detaching, any mutable second class objects (Collections, Maps, Dates etc) are typically detached as the basic form (so you can use them on client-side of your application). This property allows you to select to detach as wrapped objects. It only works with "detachAllOnCommit" situations (not with detachCopy) currently {true, **false**}
datanucleus.DetachOnClose	This allows the user to specify whether, when an EM is closed, that all objects in the L1 cache are automatically detached. **Users are recommended to use the *datanucleus.DetachAllOnCommit* wherever possible**. This will not work in JCA mode. {true, **false**}
datanucleus.detachmentFields	When detaching you can control what happens to loaded/unloaded fields of the FetchPlan. The default is to load any unloaded fields of the current FetchPlan before detaching. You can also unload any loaded fields that are not in the current FetchPlan (so you only get the fields you require) as well as a combination of both options {**load-fields**, unload-fields, load-unload-fields}
datanucleus.maxFetchDepth	Specifies the default maximum fetch depth to use for fetching operations. The Jakarta Persistence spec doesn't provide fetch group control, just a "default fetch group" type concept, consequently the default there is -1 currently. {**-1**, 1, positive integer}
datanucleus.detachedState	Allows control over which mechanism to use to determine the fields to be detached. By default DataNucleus uses the defined "fetch-groups". Obviously Jakarta Persistence doesn't have that (although it is an option with DataNucleus), so we also allow *loaded* which will detach just the currently loaded fields, and *all* which will detach all fields of the object (**be careful with this option since it, when used with maxFetchDepth of -1 will detach a whole object graph!**) {**fetch-groups**, all, loaded}
datanucleus.ServerTimeZoneID	Id of the TimeZone under which the datastore server is running. If this is not specified or is set to null it is assumed that the datastore server is running in the same timezone as the JVM under which DataNucleus is running.
datanucleus.PersistenceUnitLoadClasses	Used when we have specified the persistence-unit name for a EMF and where we want the datastore "tables" for all classes of that persistence-unit loading up into the StoreManager. Defaults to false since some databases are slow so such an operation would slow down the startup process. {true, **false**}
datanucleus.persistenceXmlFilename	URL name of the `persistence.xml` file that should be used instead of using `META-INF/persistence.xml`.
datanucleus.datastoreReadTimeout	The timeout to apply to all reads (millisecs) (query or find operations). **Only applies if the underlying datastore supports it** {**0**, positive value}
datanucleus.datastoreWriteTimeout	The timeout to apply to all writes (millisecs). (persist operations). **Only applies if the underlying datastore supports it** {**0**, positive value}

Parameter	Description + Values
datanucleus.singletonEMFForName	Whether to only allow a singleton EMF for persistence-unit. If a subsequent request is made for an EMF with a name that already exists then a warning will be logged and the original EMF returned. {true, **false**}
datanucleus.jmxType	Which JMX server to use when hooking into JMX. Please refer to the Monitoring Guide {platform}
datanucleus.type.wrapper.basis	Whether to use the "instantiated" type of a field, or the "declared" type of a field to determine which wrapper to use when the field is SCO mutable. {**instantiated**, declared}
datanucleus.type.treatJavaUtilDateAsMutable	Whether to treat java.util.Date and subtypes as mutable (and hence wrapped by a proxy). If you dont intend on calling *setTime()* on the object then setting this to false will give an efficiency benefit. {**true**, false}
datanucleus.deletionPolicy	Allows the user to decide the policy when deleting objects. The default is "JDO2" which firstly checks if the field is dependent and if so deletes dependents, and then for others will null any foreign keys out. The problem with this option is that it takes no account of whether the user has also defined foreign-key metadata, so we provide a "DataNucleus" mode that does the dependent field part first and then if a FK element is defined will leave it to the FK in the datastore to perform any actions, and otherwise does the nulling. {**JDO2**, DataNucleus}
datanucleus.identityStringTranslatorType	You can allow identities input to *em.find(id)* be translated into valid ids if there is a suitable translator. See Identity String Translator 
datanucleus.identityKeyTranslatorType	You can allow identities input to *em.find(cls, key)* be translated into valid ids if there is a suitable key translator. See Identity Key Translator 
datanucleus.datastoreIdentityType	Which "datastore-identity" class plugin to use to represent datastore identities. See Datastore Identity  {**datanucleus**, kodo, xcalia, ...}
datanucleus.executionContext.maxIdle	Specifies the maximum number of ExecutionContext objects that are pooled ready for use {**20**}
datanucleus.executionContext.reaperThread	Whether to start a reaper thread that continually monitors the pool of ExecutionContext objects and frees them off after they have surpassed their expiration period {true, **false**}
datanucleus.executionContext.closeActiveTxAction	Defines the action if an EM is closed and there is an active transaction present. {rollback, **exception**}

Parameter	Description + Values
datanucleus.stateManager.className	Class name for the StateManager to use when managing object state. The default for RDBMS is ReferentialStateManagerImpl, and is StateManagerImpl for all other datastores.
datanucleus.manageRelationships	This allows the user control over whether DataNucleus will try to manage bidirectional relations, correcting the input objects so that all relations are consistent. This process runs when flush()/commit() is called. {true, **false**}
datanucleus.manageRelationshipsChecks	This allows the user control over whether DataNucleus will make consistency checks on bidirectional relations. If "datanucleus.managedRelationships" is not selected then no checks are performed. If a consistency check fails at flush()/commit() then an exception is thrown. {true, **false**}
datanucleus.persistenceByReachabilityAtCommit	Whether to run the "persistence-by-reachability" algorithm at commit() time. This means that objects that were reachable at a call to makePersistent() but that are no longer persistent will be removed from persistence. Turn this off for performance. {true, **false**}
datanucleus.classLoaderResolverName	Name of a ClassLoaderResolver to use in class loading. This property allows the user to override the default with their own class better suited to their own loading requirements. {**datanucleus**, {name of class-loader-resolver plugin}}
datanucleus.primaryClassLoader	Sets a primary classloader for situations where a primary classloader is not accessible. This ClassLoader is used when the class is not found in the default ClassLoader search path. As example, when the database driver is loaded by a different ClassLoader not in the ClassLoader search path for Jakarta specifications.
datanucleus.plugin.pluginRegistryClassName	Name of a class that acts as registry for plug-ins. This defaults to *org.datanucleus.plugin.NonManagedPluginRegistry* (for when not using OSGi). If you are within an OSGi environment you can set this to *org.datanucleus.plugin.OSGiPluginRegistry*
datanucleus.plugin.pluginRegistryBundleCheck	Defines what happens when plugin bundles are found and are duplicated {**exception**, log, none}
datanucleus.plugin.allowUserBundles	Defines whether user-provided bundles providing DataNucleus extensions will be registered. This is only respected if used in a non-Eclipse OSGi environment. {**true**, false}
datanucleus.plugin.validatePlugins	Defines whether a validation step should be performed checking for plugin dependencies etc. This is only respected if used in a non-Eclipse OSGi environment. {true, **false**}
datanucleus.findObject.validateWhenCached	When a user calls em.find this turns off of validation when an object is found in the (L2) cache. {true, **false**}

Parameter	Description + Values
datanucleus.findObject.typeConversion	When calling em.find(Class, Object) the second argument really ought to be the exact type of the primary-key field. This property enables conversion of basic numeric types (Long, Integer, Short) to the appropriate numeric type (if the PK is a numeric type). Set this to *false* if you want strict Jakarta compliance. {**true**, false}

DataNucleus Schema Properties

DataNucleus provides the following properties for configuring schema handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.schema.autoCreateAll	Whether to automatically generate any schema, tables, columns, constraints that don't exist. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.autoCreateDatabase	Whether to automatically generate any database (catalog/schema) that doesn't exist. This depends very much on whether the datastore in question supports this operation. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.autoCreateTables	Whether to automatically generate any tables that don't exist. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.autoCreateColumns	Whether to automatically generate any columns that don't exist. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.autoCreateConstraints	Whether to automatically generate any constraints that don't exist. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.autoCreateWarnOnError	Whether to only log a warning when errors occur during the auto-creation/validation process. **Please use with care since if the schema is incorrect errors will likely come up later and this will postpone those error checks til later, when it may be too late!!** {true, **false**}
datanucleus.schema.validateAll	Alias for defining **datanucleus.schema.validateTables**, **datanucleus.schema.validateColumns** and **datanucleus.schema.validateConstraints** as all true. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.validateTables	Whether to validate tables against the persistence definition. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.validateColumns	Whether to validate columns against the persistence definition. This refers to the column detail structure and NOT to whether the column exists or not. Please refer to the Schema Guide for more details. {true, **false**}

Parameter	Description + Values
datanucleus.schema.validateConstraints	Whether to validate table constraints against the persistence definition. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.readOnlyDatastore	Whether the datastore is read-only or not (fixed in structure and contents) {true, **false**}
datanucleus.readOnlyDatastoreAction	What happens when a datastore is read-only and an object is attempted to be persisted. {**exception**, ignore}
datanucleus.schema.generateDatabase.mode	Whether to perform any schema generation to the database at startup. Will process the schema for all classes that have metadata loaded at startup (i.e the classes specified in a persistence-unit). {create, drop, drop-and-create, **none**}
datanucleus.schema.generateScripts.mode	Whether to perform any schema generation into scripts at startup. Will process the schema for all classes that have metadata loaded at startup (i.e the classes specified in a persistence-unit). {create, drop, drop-and-create, **none**}
datanucleus.schema.generateScripts.create	Name of the script file to write to if doing a "create" with the target as "scripts" {**datanucleus-schema-create.ddl**, {filename}}
datanucleus.schema.generateScripts.drop	Name of the script file to write to if doing a "drop" with the target as "scripts" {**datanucleus-schema-drop.ddl**, {filename}}
datanucleus.schema.generateDatabase.createScript	Name of a script file to run to create tables. Can be absolute filename, or URL string
datanucleus.schema.generateDatabase.dropScript	Name of a script file to run to drop tables. Can be absolute filename, or URL string
datanucleus.schema.loadScript	Name of a script file to run to load data into the schema. Can be absolute filename, or URL string
datanucleus.identifierFactory	Name of the identifier factory to use when generating table/column names etc (RDBMS datastores only). See also the Datastore Identifier Guide. {datanucleus1, datanucleus2, jpox, jpa, **jakarta**, {user-plugin-name}}
datanucleus.identifier.namingFactory	Name of the identifier NamingFactory to use when generating table/column names etc (non-RDBMS datastores). {datanucleus2, jpa, **jakarta**, {user-plugin-name}}
datanucleus.identifier.case	Which case to use in generated table/column identifier names. See also the Datastore Identifier Guide RDBMS defaults to UPPERCASE. Cassandra defaults to lowercase {UPPERCASE, lowercase, MixedCase}
datanucleus.identifier.wordSeparator	Separator character(s) to use between words in generated identifiers. Defaults to "_" (underscore)
datanucleus.identifier.tablePrefix	Prefix to be prepended to all generated table names (if the identifier factory supports it)

Parameter	Description + Values
datanucleus.identifier.tableSuffix	Suffix to be appended to all generated table names (if the identifier factory supports it)
datanucleus.store.allowReferencesWithNoImplementations	Whether we permit a reference field (1-1 relation) or collection of references where there are no defined implementations of the reference. False means that an exception will be thrown during schema generation for the field {true, **false**}

DataNucleus Transaction Properties

DataNucleus provides the following properties for configuring transaction handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.transaction.type	Type of transaction to use. If running under JavaSE the default is RESOURCE_LOCAL, and if running under JavaEE the default is JTA. {RESOURCE_LOCAL, JTA}
datanucleus.transaction.isolation	Select the default transaction isolation level for ALL EntityManagers. Some databases do not support all isolation levels, refer to your database documentation. Please refer to the transaction guide {read-uncommitted, **read-committed**, repeatable-read, serializable}
datanucleus.transaction.jta.transactionManagerLocator	Selects the locator to use when using JTA transactions so that DataNucleus can find the JTA TransactionManager. If this isn't specified and using JTA transactions DataNucleus will search all available locators which could have a performance impact. See JTA Locator  . If specifying "custom_jndi" please also specify "datanucleus.transaction.jta.transactionManagerJNDI" {**autodetect**, jboss, jonas, jotm, oc4j, orion, resin, sap, sun, weblogic, websphere, custom_jndi, alias of a JTA transaction locator}
datanucleus.transaction.jta.transactionManagerJNDI	Name of a JNDI location to find the JTA transaction manager from (when using JTA transactions). This is for the case where you know where it is located. If not used DataNucleus will try certain well-known locations
datanucleus.transaction.nontx.read	Whether to allow nontransactional reads {false, **true**}
datanucleus.transaction.nontx.write	Whether to allow nontransactional writes {false, **true**}

Parameter	Description + Values
datanucleus.transaction.nontx.atomic	When a user invokes a nontransactional operation they can choose for these changes to go straight to the datastore (atomically) or to wait until either the next transaction commit, or close of the EM. Disable this if you want operations to be processed with the next real transaction. {true, **false**}
datanucleus.SerializeRead	With datastore transactions you can apply locking to objects as they are read from the datastore. This setting applies as the default for all EMs obtained. You can also specify this on a per-transaction or per-query basis (which is often better to avoid deadlocks etc) {true, **false**}
datanucleus.flush.auto.objectLimit	For use when using (DataNucleus) "AUTO" flush mode (see `datanucleus.flush.mode`) and is the limit on number of dirty objects before a flush to the datastore will be performed. {**1**, positive integer}
datanucleus.flush.mode	Sets when persistence operations are flushed to the datastore. This overrides the Jakarta flush mode. *MANUAL* means that operations will be sent only on flush()/commit() (**same as Jakarta FlushModeType.COMMIT**). *QUERY* means that operations will be sent on flush()/commit() and just before query execution (**same as Jakarta FlushModeType.AUTO**). *AUTO* means that operations will be sent immediately (auto-flush). {MANUAL, QUERY, AUTO}
datanucleus.flush.optimised	Whether to use an "optimised" flush process, changing the order of persists for referential integrity (as used by RDBMS typically), or whether to just build a list of deletes, inserts and updates and do them in batches. RDBMS defaults to true, whereas other datastores default to false (due to not having referential integrity, so gaining from batching {true, false}

DataNucleus Cache Properties

DataNucleus provides the following properties for configuring cache handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.cache.collections	SCO collections can be used in 2 modes in DataNucleus. You can allow DataNucleus to cache the collections contents, or you can tell DataNucleus to access the datastore for every access of the SCO collection. The default is to use the cached collection. {**true**, false}
datanucleus.cache.collections.lazy	When using cached collections/maps, the elements/keys/values can be loaded when the object is initialised, or can be loaded when accessed (lazy loading). The default is to use lazy loading when the field is not in the current fetch group, and to not use lazy loading when the field is in the current fetch group. {true, false}

Parameter	Description + Values
datanucleus.cache.level1.type	Name of the type of Level 1 cache to use. Defines the backing map. See also Cache docs for Jakarta {**soft**, weak, strong, {your-plugin-name}}
datanucleus.cache.level2.type	Name of the type of Level 2 Cache to use. Can be used to interface with external caching products. Use "none" to turn off L2 caching. See also Cache docs for Jakarta {none, **soft**, weak, javax.cache, coherence, ehcache, ehcacheclassbased, redis, cacheonix, oscache, spymemcached, xmemcached, {your-plugin-name}
datanucleus.cache.level2.mode	The mode of operation of the L2 cache, deciding which entities are cached. The default (UNSPECIFIED) is the same as DISABLE_SELECTIVE. See also Cache docs for Jakarta {NONE, ALL, ENABLE_SELECTIVE, DISABLE_SELECTIVE, **UNSPECIFIED**}
datanucleus.cache.level2.storeMode	Whether to use the L2 cache for storing values (set to "bypass" to not store within the context of the operation) {**use**, bypass}
datanucleus.cache.level2.retrieveMode	Whether to use the L2 cache for retrieving values (set to "bypass" to not retrieve from L2 cache within the context of the operation, i.e go to the datastore) {**use**, bypass}
datanucleus.cache.level2.updateMode	When the objects in the L2 cache should be updated. Defaults to updating at commit AND when fields are read from a datastore object {**commit-and-datastore-read**, commit-only, datastore-read-only}
datanucleus.cache.level2.cacheName	Name of the cache. This is for use with plugins such as the Tangosol cache plugin for accessing the particular cache. Please refer to the L2 Cache docs
datanucleus.cache.level2.maxSize	Max size for the L2 cache (supported by weak, soft, coherence, ehcache, ehcacheclassbased, javax.cache) {**-1**, integer value}
datanucleus.cache.level2.clearAtClose	Whether the close of the L2 cache (when the EMF closes) should also clear out any objects from the underlying cache mechanism. By default it will clear objects out but if the user has configured an external cache product and wants to share objects across multiple EMFs then this can be set to false. {**true**, false}
datanucleus.cache.level2.batchSize	When objects are added to the L2 cache at commit they are typically batched. This property sets the max size of the batch. {**100**, integer value}
datanucleus.cache.level2.expiryMillis	Some caches (Cacheonix, Redis) allow specification of an expiration time for objects in the cache. This property is the timeout in milliseconds (will be unset meaning use cache default). {**-1**, integer value}
datanucleus.cache.level2.readThrough	With javax.cache L2 caches you can configure the cache to allow read-through {**true**, false}
datanucleus.cache.level2.writeThrough	With javax.cache L2 caches you can configure the cache to allow write-through {**true**, false}
datanucleus.cache.level2.storeByValue	With javax.cache L2 caches you can configure the cache to store by value (as opposed to by reference) {**true**, false}

Parameter	Description + Values
datanucleus.cache.level2.statisticsEnabled	With javax.cache L2 caches you can configure the cache to enable statistics gathering (accessible via JMX) {**false**, true}
datanucleus.cache.queryCompilation.type	Type of cache to use for caching of generic query compilations {none, **soft**, weak, strong, javax.cache, {your-plugin-name}}
datanucleus.cache.queryCompilation.cacheName	Name of cache for generic query compilation. Used by javax.cache variant. {{your-cache-name}, **datanucleus-query-compilation**}
datanucleus.cache.queryCompilationDatastore.type	Type of cache to use for caching of datastore query compilations {none, **soft**, weak, strong, javax.cache, {your-plugin-name}}
datanucleus.cache.queryCompilationDatastore.cacheName	Name of cache for datastore query compilation. Used by javax.cache variant. {{your-cache-name}, **datanucleus-query-compilation-datastore**}
datanucleus.cache.queryResults.type	Type of cache to use for caching query results. {none, **soft**, weak, strong, javax.cache, redis, spymemcached, xmemcached, cacheonix, {your-plugin-name}}
datanucleus.cache.queryResults.cacheName	Name of cache for caching the query results. {**datanucleus-query**, {your-name}}
datanucleus.cache.queryResults.clearAtClose	Whether the close of the Query Results cache (when the EMF closes) should also clear out any objects from the underlying cache mechanism. By default it will clear query results out. {**true**, false}
datanucleus.cache.queryResults.maxSize	Max size for the query results cache (supported by weak, soft, strong) {**-1**, integer value}
datanucleus.cache.queryResults.expiryMillis	Expiry in milliseconds for objects in the query results cache (cacheonix, redis) {**-1**, integer value}

DataNucleus Bean Validation Properties

DataNucleus provides the following properties for configuring bean validation handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.validation.mode	Determines whether the automatic lifecycle event validation is in effect. {**auto**, callback, none}
datanucleus.validation.group.pre-persist	The classes to validation on pre-persist callback
datanucleus.validation.group.pre-update	The classes to validation on pre-update callback

Parameter	Description + Values
datanucleus.validation.group.pre-remove	The classes to validation on pre-remove callback
datanucleus.validation.factory	The validation factory to use in validation

DataNucleus Value Generation Properties

DataNucleus provides the following properties for configuring value generation handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.valuegeneration.transactionAttribute	Whether to use the EM connection or open a new connection. Only used by value generators that require a connection to the datastore. {**NEW**, EXISTING}
datanucleus.valuegeneration.transactionIsolation	Select the default transaction isolation level for identity generation. Must have *datanucleus.valuegeneration.transactionAttribute* set to *New*. Some databases do not support all isolation levels, refer to your database documentation. Please refer to the transaction guide {read-uncommitted, **read-committed**, repeatable-read, serializable}

DataNucleus Metadata Properties

DataNucleus provides the following properties for configuring metadata handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.metadata.alwaysDetachable	Whether to treat all classes as detachable irrespective of input metadata. See also "alwaysDetachable" enhancer option. {**false**, true}
datanucleus.metadata.listener.object	Property specifying a org.datanucleus.metadata.MetaDataListener object that will be registered at startup and will receive notification of all metadata load activity. {**false**, true}
datanucleus.metadata.ignoreMetaDataForMissingClasses	Whether to ignore classes where metadata is specified. Default (false) is to throw an exception. {**false**, true}
datanucleus.metadata.xml.validate	Whether to validate the MetaData file(s) for XML correctness (against the DTD) when parsing. {true, **false**}

Parameter	Description + Values
datanucleus.metadata.xml.namespaceAware	Whether to allow for XML namespaces in metadata files. The vast majority of sane people should not need this at all, but it's enabled by default to allow for those that do. {**true**, false}
datanucleus.metadata.allowXML	Whether to allow XML metadata. Turn this off if not using any, for performance. {**true**, false}
datanucleus.metadata.allowAnnotations	Whether to allow annotations metadata. Turn this off if not using any, for performance. {**true**, false}
datanucleus.metadata.allowLoadAtRuntime	Whether to allow load of metadata at runtime. This is intended for the situation where you are handling persistence of a persistence-unit and only want the classes explicitly specified in the persistence-unit. {**true**, false}
datanucleus.metadata.defaultNullable	Whether the default nullability for the fields should be nullable or non-nullable when no metadata regarding field nullability is specified at field level. The default is nullable i.e. to allow null values (since v5.0.0). {**true**, false}
datanucleus.metadata.scanner	Name of a class to use for scanning the classpath for persistent classes when using a `persistence.xml`. The class must implement the interface *org.datanucleus.metadata.MetaDataScanner*
datanucleus.metadata.useDiscriminatorForSingleTable	With Jakarta Persistence the spec implies that all use of "single-table" inheritance will use a discriminator. DataNucleus up to and including 5.0.2 relied on the user defining the discriminator, whereas it now will add one if not supplied. Set this to *false* to get behaviour as it was ⇐ 5.0.2 {**true**, false}
datanucleus.metadata.javaxValidationShortcuts	Whether to process javax.validation `@NotNull` and `@Size` annotations as their Jakarta `@Column` equivalent. {**false**, true}

DataNucleus Query Properties

DataNucleus provides the following properties for configuring query handling used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.query.flushBeforeExecution	This property can enforce a flush to the datastore of any outstanding changes just before executing all queries. If using optimistic locking any updates are typically held back until flush/commit and so the query would otherwise not take them into account. {true, **false**}
datanucleus.query.jpql.allowRange	JPQL queries, by the Jakarta Persistence spec, do not allow specification of the range in the query string. This extension to allow "RANGE x,y" after the ORDER BY clause of JPQL string queries. {**false**, true}

Parameter	Description + Values
datanucleus.query.checkUnusedParameters	Whether to check for unused input parameters and throw an exception if found. The Jakarta Persistence spec requires this check and is a good guide to having misnamed a parameter name in the query for example. {**true**, false}
datanucleus.query.sql.syntaxChecks	Whether to perform some basic syntax checking on SQL/"native" queries that they include PK, version and discriminator columns where necessary. {**true**, false}

DataNucleus Datastore-Specific Properties

DataNucleus provides the following properties for configuring datastore-specific used by the EntityManagerFactory.

Parameter	Description + Values
datanucleus.rdbms.datastoreAdapterClassName	This property allows you to supply the class name of the adapter to use for your datastore. The default is not to specify this property and DataNucleus will autodetect the datastore type and use its own internal datastore adapter classes. This allows you to override the default behaviour where there maybe is some issue with the default adapter class. **Applicable for RDBMS only**
datanucleus.rdbms.useLegacyNativeValueStrategy	This property changes the process for deciding the value strategy to use when the user has selected "auto" to be like it was with version 3.0 and earlier, so using "increment" and "uuid-hex". **Applicable for RDBMS only** {true, **false**}
datanucleus.rdbms.statementBatchLimit	Maximum number of statements that can be batched. The default is 50 and also applies to delete of objects. Please refer to the Statement Batching guide **Applicable for RDBMS only** {integer value (0 = no batching)}
datanucleus.rdbms.checkExistTablesOrViews	Whether to check if the table/view exists. If false, it disables the automatic generation of tables that don't exist. **Applicable for RDBMS only** {**true**, false}
datanucleus.rdbms.useDefaultSqlType	This property applies for schema generation in terms of setting the default column "sql-type" (when you haven't defined it) and where the JDBC driver has multiple possible "sql-type" for a "jdbc-type". If the property is set to false, it will take the first provided "sql-type" from the JDBC driver. If the property is set to true, it will take the "sql-type" that matches what the DataNucleus "plugin.xml" implies. **Applicable for RDBMS only**. {**true**, false}

Parameter	Description + Values
datanucleus.rdbms.initializeColumnInfo	Allows control over what column information is initialised when a table is loaded for the first time. By default info for all columns will be loaded. Unfortunately some RDBMS are particularly poor at returning this information so we allow reduced forms to just load the primary key column info, or not to load any. **Applicable for RDBMS only** {**ALL**, PK, NONE}
datanucleus.rdbms.classAdditionMaxRetries	The maximum number of retries when trying to find a class to persist or when validating a class. **Applicable for RDBMS only** {**3**, A positive integer}
datanucleus.rdbms.constraintCreateMode	How to determine the RDBMS constraints to be created. **DataNucleus** will automatically add foreign-keys/indices to handle all relationships, and will utilise the specified MetaData foreign-key information. **JDO2** will only use the information in the MetaData file(s). **Applicable for RDBMS only**. {**DataNucleus**, JDO2}
datanucleus.rdbms.uniqueConstraints.mapInverse	Whether to add unique constraints to the element table for a map inverse (FK) field. **Applicable for RDBMS only**. {**true**, false}
datanucleus.rdbms.discriminatorPerSubclassTable	Property that controls if only the base class where the discriminator is defined will have a discriminator column **Applicable for RDBMS only**. {**false**, true}
datanucleus.rdbms.stringDefaultLength	The default (max) length to use for all strings that don't have their column length defined in MetaData. **Applicable for RDBMS only**. {**255**, A valid length}
datanucleus.rdbms.stringLengthExceededAction	Defines what happens when persisting a String field and its length exceeds the length of the underlying datastore column. The default is to throw an Exception. The other option is to truncate the String to the length of the datastore column. **Applicable for RDBMS only** {**EXCEPTION**, TRUNCATE}
datanucleus.rdbms.useColumnDefaultWhenNull	If an object is being persisted and a field (column) is null, the default behaviour is to look whether the column has a "default" value defined in the datastore and pass that in. You can turn this off and instead pass in NULL for the column by setting this property to *false*. **Applicable for RDBMS only**. {**true**, false}
datanucleus.rdbms.persistEmptyStringAsNull	When persisting an empty string, should it be persisted as null in the datastore? This is to allow for datastores such as Oracle that dont differentiate between null and empty string. If it is set to false and the datastore doesnt differentiate then a special character will be saved when storing an empty string (and interpreted when reading in). **Applicable for RDBMS only** {true, **false**}
datanucleus.rdbms.query.fetchDirection	The direction in which the query results will be navigated. **Applicable for RDBMS only** {**forward**, reverse, unknown}

Parameter	Description + Values
datanucleus.rdbms.query.resultSetType	Type of ResultSet to create. Note 1) Not all JDBC drivers accept all options. The values correspond directly to the ResultSet options. Note 2) Not all java.util.List operations are available for scrolling result sets. An Exception is raised when unsupported operations are invoked. **Applicable for RDBMS only**. {**forward-only**, scroll-sensitive, scroll-insensitive}
datanucleus.rdbms.query.resultSetConcurrency	Whether the ResultSet is readonly or can be updated. Not all JDBC drivers support all options. The values correspond directly to the ResultSet options. **Applicable for RDBMS only** {**read-only**, updateable}
datanucleus.rdbms.query.multivaluedFetch	How any multi-valued field should be fetched in a query. 'exists' means use an EXISTS statement hence retrieving all elements for the queried objects in one SQL with EXISTS to select the affected owner objects. 'none' means don't fetch container elements. **Applicable for RDBMS only** {**exists**, none}
datanucleus.rdbms.oracle.nlsSortOrder	Sort order for Oracle String fields in queries (BINARY disables native language sorting). **Applicable to Oracle only** {**LATIN**, See Oracle documentation}
datanucleus.rdbms.mysql.engineType	Specify the default engine for any tables created in MySQL. **Applicable to MySQL only**. {**InnoDB**, valid engine for MySQL}
datanucleus.rdbms.mysql.collation	Specify the default collation for any tables created in MySQL. **Applicable to MySQL only**
datanucleus.rdbms.mysql.characterSet	Specify the default charset for any tables created in MySQL. **Applicable to MySQL only**
datanucleus.rdbms.informix.useSerialForIdentity	Whether we are using SERIAL for identity columns (instead of SERIAL8). **Applicable to Informix only.** {true, **false**}
datanucleus.rdbms.dynamicSchemaUpdates	Whether to allow dynamic updates to the schema. This means that upon each insert/update the types of objects will be tested and any previously unknown implementations of interfaces will be added to the existing schema. **Applicable for RDBMS only** {true, **false**}
datanucleus.rdbms.omitDatabaseMetaDataGetColumns	Whether to bypass all calls to DatabaseMetaData.getColumns(). This JDBC method is called to get schema information, but on some JDBC drivers (e.g Derby) it can take an inordinate amout of time. Setting this to true means that your datastore schema has to be correct and no checks will be performed. **Applicable for RDBMS only**. {true, **false**}
datanucleus.rdbms.sqlTableNamingStrategy	Name of the plugin to use for defining the names of the aliases of tables in SQL statements. **Applicable for RDBMS only** {**alpha-scheme**, t-scheme}

Parameter	Description + Values
datanucleus.rdbms.tableColumnOrder	How we should order the columns in a table. The default is to put the fields of the owning class first, followed by superclasses, then subclasses. An alternative is to start from the base superclass first, working down to the owner, then the subclasses **Applicable for RDBMS only**. {**owner-first**, superclass-first}
datanucleus.rdbms.allowColumnReuse	This property allows you to reuse columns for more than 1 field of a class. It is *false* by default to protect the user from erroneously typing in a column name. Additionally, if a column is reused, the user ought to think about how to determine which field is written to that column … all reuse ought to imply the same value in those fields so it doesn't matter which field is written there, or retrieved from there. **Applicable for RDBMS only** {true, **false**}
datanucleus.rdbms.statementLogging	How to log SQL statements. The default is to log the raw JDBC statement (with ? for parameters). Alternatively you can log the statement with any parameters replaced by just the values (no brackets). The final option is to log the statement and replace any parameters with the value provided in angle brackets. **Applicable for RDBMS only** {**JDBC**, PARAMS_INLINE, PARAMS_IN_BRACKETS}
datanucleus.rdbms.fetchUnloadedAutomatically	If enabled will, upon a request to load a field, check for any unloaded fields that are non-relation fields or 1-1/N-1 fields and will load them in the same SQL call. **Applicable for RDBMS only** {true, **false**}
datanucleus.cloud.storage.bucket	This is a mandatory property that allows you to supply the bucket name to store your data. **Applicable for Google Storage, and AmazonS3 only.**
datanucleus.hbase.relationUsesPersistableId	This defines how relations will be persisted. The legacy method would be just to store the "id" of the object. The default method is to use "persistableId" which is a form of the id but catering for datastore id and application id, and including the class of the target object to avoid subsequent lookups. **Applicable for HBase only.** {**true**, false}
datanucleus.hbase.enforceUniquenessInApplication	Setting this property to true means that when a new object is persisted (and its identity is assigned), no check will be made as to whether it exists in the datastore and that the user takes responsibility for such checks. **Applicable for HBase only.** {true, **false**}
datanucleus.cassandra.enforceUniquenessInApplication	Setting this property to true means that when a new object is persisted (and its identity is assigned), no check will be made as to whether it exists in the datastore (since Cassandra does an UPSERT) and that the user takes responsibility for such checks. **Applicable for Cassandra only.** {true, **false**}
datanucleus.cassandra.compression	Type of compression to use for the Cassandra cluster. **Applicable for Cassandra only.** {**none**, snappy}
datanucleus.cassandra.metrics	Whether metrics are enabled for the Cassandra cluster. **Applicable for Cassandra only.** {**true**, false}

Parameter	Description + Values
datanucleus.cassandra.ssl	Whether SSL is enabled for the Cassandra cluster. **Applicable for Cassandra only.** {true, **false**}
datanucleus.cassandra.socket.readTimeoutMillis	Socket read timeout for the Cassandra cluster. **Applicable for Cassandra only.**
datanucleus.cassandra.socket.connectTimeoutMillis	Socket connect timeout for the Cassandra cluster. **Applicable for Cassandra only.**
datanucleus.cassandra.loadBalancingPolicy	Sets the load balancing policy to use. **Applicable for Cassandra only.** {round-robin, token-aware}
datanucleus.cassandra.loadBalancingPolicy.tokenAwareLocalDC	Sets the local DC to use for the load balancing policy. **Applicable for Cassandra only.**

DataNucleus EMF Properties

DataNucleus provides the following properties for configuring EMF capabilities.

Parameter	Description + Values
datanucleus.jakarta.addClassTransformer	When running with Jakarta Persistence in a JavaEE environment if you wish to have your classes enhanced at runtime you can enable this by setting this property to *true*. The default is to bytecode enhance your classes before deployment. {**false**, true}
datanucleus.jakarta.persistenceContextType	Jakarta Persistence defines two lifecycle options. JavaEE usage defaults to "transaction" where objects are detached when a transaction is committed. JavaSE usage defaults to "extended" where objects are detached when the EntityManager is closed. This property allows control {transaction, extended}
datanucleus.jakarta.txnMarkForRollbackOnException	Jakarta Persistence requires that any persistence exception should mark the current transaction for rollback. This persistence property allows that inflexible behaviour to be turned off leaving it to the user to decide when a transaction is needing to be rolled back. {**true**, false}

Closing EntityManagerFactory

Since the EMF has significant resources associated with it, it should always be closed when you no longer need to perform any more persistence operations. For most operations this will be when closing your application. Whenever it is you do it like this

```
emf.close();
```

Level 2 Cache

The *EntityManagerFactory* has an optional cache of all objects across all _EntityManager_s. This cache is called the **Level 2 (L2) cache**, and Jakarta Persistence doesn't define whether this should be enabled or not. With DataNucleus it defaults to enabled. The user can configure the L2 cache if they so wish; by use of the persistence property **datanucleus.cache.level2.type**. You set this to "type" of cache required. You currently have the following options.

- **soft** - use the internal (soft reference based) L2 cache. **This is the default L2 cache in DataNucleus.** Provides support for the Jakarta Persistence interface of being able to put objects into the cache, and evict them when required. This option does not support distributed caching, solely running within the JVM of the client application. Soft references are held to non pinned objects.
- **weak** - use the internal (weak reference based) L2 cache. Provides support for the Jakarta Persistence interface of being able to put objects into the cache, and evict them when required. This option does not support distributed caching, solely running within the JVM of the client application. Weak references are held to non pinned objects.
- javax.cache - a simple wrapper to the Java standard "javax.cache" Temporary Caching API.
- EHCache - a simple wrapper to EHCache's caching product.
- EHCacheClassBased - similar to the EHCache option but class-based.
- Redis - a simple L2 cache using Redis.
- Oracle Coherence - a simple wrapper to Oracle's Coherence caching product. Oracle's caches support distributed caching, so you could, in principle, use DataNucleus in a distributed environment with this option.
- spymemcached - a simple wrapper to the "spymemcached" client for memcached caching product.
- xmemcached - a simple wrapper to the "xmemcached" client for memcached caching product.
- cacheonix - a simple wrapper to the Cacheonix distributed caching software.
- OSCache - a simple wrapper to OSCache's caching product.
- **none** - turn OFF L2 caching.

The weak, soft and javax.cache caches are available in the datanucleus-core plugin. The EHCache, OSCache, Coherence, Cacheonix, and Memcache caches are available in the datanucleus-cache plugin.

In addition you can control the *mode* of operation of the L2 cache. You do this using the persistence property **datanucleus.cache.level2.mode** (or **jakarta.persistence.sharedCache.mode**). The default is *UNSPECIFIED* which means that DataNucleus will cache all objects of entities unless the entity is explicitly marked as not cacheable. The other options are *NONE* (don't cache ever), *ALL* (cache all entities regardless of annotations), *ENABLE_SELECTIVE* (cache entities explicitly marked

as cacheable), or *DISABLE_SELECTIVE* (cache entities unless explicitly marked as not cacheable - i.e same as our default).

Objects are placed in the L2 cache when you commit() the transaction of a EntityManager. This means that you only have datastore-persisted objects in that cache. Also, if an object is deleted during a transaction then at commit it will be removed from the L2 cache if it is present.

The L2 cache is a DataNucleus 

allowing you to provide your own cache where you require it. Use the examples of the EHCache, Coherence caches etc as reference.

Controlling the Level 2 Cache

The majority of times when using a Jakarta Persistence-enabled system you will not have to take control over any aspect of the caching other than specification of whether to use a **L2 Cache** or not. With Jakarta Persistence and DataNucleus you have the ability to control which objects remain in the cache. This is available via a method on the *EntityManagerFactory*.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(persUnitName,
props);
Cache cache = emf.getCache();
```

The *Cache* interface provides methods to control the retention of objects in the cache. You have 2 types of methods

- **contains** - check if an object of a type with a particular identity is in the cache
- **evict** - used to remove objects from the Level 2 Cache

You can also control which classes are put into a Level 2 cache. So with the following Jakarta annotation `@Cacheable`, no objects of type *MyClass* will be put in the L2 cache.

```
@Cacheable(false)
@Entity
public class MyClass
{
    ...
}
```

If you want to control which fields of an object are put in the Level 2 cache you can do this using an extension annotation on the field. This setting is only required for fields that are relationships to other persistable objects. Like this

```
public class MyClass
{
    ...

    Collection values;

    @Extension(vendorName="datanucleus", key="cacheable", value="false")
    Collection elements;
}
```

So in this example we will cache "values" but not "elements". If a field is *cacheable* then

- If it is a persistable object, the "identity" of the related object will be stored in the Level 2 cache for this field of this object
- If it is a Collection of persistable elements, the "identity" of the elements will be stored in the Level 2 cache for this field of this object
- If it is a Map of persistable keys/values, the "identity" of the keys/values will be stored in the Level 2 cache for this field of this object

When pulling an object in from the Level 2 cache and it has a reference to another object DataNucleus uses the "identity" to find that object in the Level 1 or Level 2 caches to re-relate the objects.

L2 Cache using javax.cache

DataNucleus provides a simple wrapper to any compliant javax.cache implementation, for example Apache Ignite or HazelCast. To enable this you should put a "javax.cache" implementation in your CLASSPATH, and set the persistence properties

```
datanucleus.cache.level2.type=javax.cache
datanucleus.cache.level2.cacheName={cache name}
```

As an example, you could simply add the following to a Maven POM, together with those persistence properties above to use HazelCast "javax.cache" implementation

```
<dependency>
    <groupId>javax.cache</groupId>
    <artifactId>cache-api</artifactId>
    <version>1.0.0</version>
</dependency>
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast</artifactId>
    <version>3.7.3</version>
</dependency>
```

L2 Cache using EHCache

DataNucleus provides a simple wrapper to EHCache's own API caches (not the javax.cache API variant). To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=ehcache
datanucleus.cache.level2.cacheName={cache name}
datanucleus.cache.level2.configurationFile={EHCache configuration file (in classpath)}
```

The EHCache plugin also provides an alternative L2 Cache that is class-based. To use this you would need to replace "ehcache" above with "ehcacheclassbased".

L2 Cache using Spymemcached/Xmemcached

DataNucleus provides a simple wrapper to Spymemcached caches and Xmemcached caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=spymemcached         [or "xmemcached"]
datanucleus.cache.level2.cacheName={prefix for keys, to avoid clashes with other
memcached objects}
datanucleus.cache.level2.memcached.servers=...
datanucleus.cache.level2.expireMillis=...
```

datanucleus.cache.level2.memcached.servers is a space separated list of memcached hosts/ports, e.g. host:port host2:port. **datanucleus.cache.level2.expireMillis** if not set or set to 0 then no expire

L2 Cache using Cacheonix

DataNucleus provides a simple wrapper to Cacheonix. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=cacheonix
datanucleus.cache.level2.cacheName={cache name}
```

Note that you can optionally also specify

```
datanucleus.cache.level2.expiryMillis={timeout-in-millis (default=60)}
datanucleus.cache.level2.configurationFile={Cacheonix configuration file (in
classpath)}
```

and define a *cacheonix-config.xml* like

```
<?xml version="1.0"?>
<cacheonix>
   <local>
      <!-- One cache per class being stored. -->
      <localCache name="mydomain.MyClass">
         <store>
            <lru maxElements="1000" maxBytes="1mb"/>
            <expiration timeToLive="60s"/>
         </store>
      </localCache>

      <!-- Fallback cache for classes indeterminable from their id. -->
      <localCache name="datanucleus">
         <store>
            <lru maxElements="1000" maxBytes="10mb"/>
            <expiration timeToLive="60s"/>
         </store>
      </localCache>

      <localCache name="default" template="true">
         <store>
            <lru maxElements="10" maxBytes="10mb"/>
            <overflowToDisk maxOverflowBytes="1mb"/>
            <expiration timeToLive="1s"/>
         </store>
      </localCache>
   </local>

</cacheonix>
```

L2 Cache using Redis

DataNucleus provides a simple L2 cache using Redis. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=redis
datanucleus.cache.level2.cacheName={cache name}
datanucleus.cache.level2.clearAtClose={true | false, whether to clear at close}
datanucleus.cache.level2.expireMillis=...
datanucleus.cache.level2.redis.database={database, or use the default '1'}
datanucleus.cache.level2.redis.timeout={optional cache timeout, or use the default of
5000}
datanucleus.cache.level2.redis.sentinels={comma-separated list of sentinels, optional
(use server/port instead)}
datanucleus.cache.level2.redis.server={server, or use the default of "localhost"}
datanucleus.cache.level2.redis.port={port, or use the default of 6379}
```

L2 Cache using OSCache

DataNucleus provides a simple wrapper to OSCache's caches. To enable this you should set the persistence properties

```
datanucleus.cache.level2.type=oscache
datanucleus.cache.level2.cacheName={cache name}
```

L2 Cache using Oracle Coherence

DataNucleus provides a simple wrapper to Oracle's Coherence caches. This currently takes the *NamedCache* interface in Coherence and instantiates a cache of a user provided name. To enabled this you should set the following persistence properties

```
datanucleus.cache.level2.type=coherence
datanucleus.cache.level2.cacheName={coherence cache name}
```

The *Coherence cache name* is the name that you would normally put into a call to CacheFactory.getCache(name). You have the benefits of Coherence's distributed/serialized caching. If you require more control over the Coherence cache whilst using it with DataNucleus, you can just access the cache directly via

```
JakartaDataStoreCache cache = (JakartaDataStoreCache)emf.getCache();
NamedCache tangosolCache = ((TangosolLevel2Cache)cache.getLevel2Cache
()).getTangosolCache();
```

Level 2 Cache implementation

Objects in a Level 2 cache are keyed by their Jakarta "identity". Consequently only persistable objects with an identity will be L2 cached. In terms of what is cached, the persistable object is represented by a CachedPC object. This stores the class of the persistable object, the "id", "version" (if present), and the field values (together with which fields are present in the L2 cache). If a field is/contains a relation, the field value will be the "id" of the related object (rather than the object itself). If a field is/contains an embedded persistable object, the field value will be a nested `CachedPC` object representing that object.

Datastore Schema

Some datastores have a well-defined structure and when persisting/retrieving from these datastores you have to have this *schema* in place. DataNucleus provides various controls for creation of any necessary schema components. This creation can be performed as follows

- At runtime, as a one-off generate-schema step. This is the recommended option since it is standard in Jakarta Persistence.
- One off task before running your application using SchemaTool
- At runtime, auto-generating tables as it requires them

The thing to remember when using DataNucleus is that **the schema is under your control**. DataNucleus does not impose anything on you as such, and you have the power to turn on/off all schema components. Some Java persistence tools add various types of information to the tables for persisted classes, such as special columns, or meta information. DataNucleus is very unobtrusive as far as the datastore schema is concerned. It minimises the addition of any implementation artifacts to the datastore, and adds *nothing* (other than any datastore identities, and version columns where requested) to any schema tables.

Schema Generation for persistence-unit

DataNucleus Jakarta allows you to generate the schema for your *persistence-unit* when creating an EMF. You can create, drop or drop then create the schema either directly in the datastore, or in scripts (DDL) as required. See the associated persistence properties (most of these only apply to RDBMS).

- **jakarta.persistence.schema-generation.database.action** which can be set to *create*, *drop*, *drop-and-create* or *none* to control the generation of the schema in the database.
- **jakarta.persistence.schema-generation.scripts.action** which can be set to *create*, *drop*, *drop-and-create* or *none* to control the generation of the schema as scripts (DDL). See also *jakarta.persistence.schema-generation.scripts.create.target* and *jakarta.persistence.schema-generation.scripts.drop.target* which will be generated using this mode of operation.
- **jakarta.persistence.schema-generation.scripts.create.target** - this should be set to the name of a DDL script file that will be generated when using *jakarta.persistence.schema-generation.scripts.action*
- **jakarta.persistence.schema-generation.scripts.drop.target** - this should be set to the name of a DDL script file that will be generated when using *jakarta.persistence.schema-generation.scripts.action*
- **jakarta.persistence.schema-generation.scripts.create.source** - set this to an SQL script of your own that will create some tables (prior to any schema generation from the persistable objects)
- **jakarta.persistence.schema-generation.scripts.drop.source** - set this to an SQL script of your own that will drop some tables (prior to any schema generation from the persistable objects)
- **jakarta.persistence.sql-load-script-source** - set this to an SQL script of your own that will

insert any data that you require to be available when your EMF is initialised

Some examples.

Example 1, to create a database using the Jakarta entities (metadata), and then load data

```
jakarta.persistence.schema-generation.database.action=create
jakarta.persistence.sql-load-script-source=/usr/local/MyStartUp.sql
```

Example 2, to create DDL scripts for the Jakarta entities

```
jakarta.persistence.schema-generation.scripts.action=create
jakarta.persistence.schema-
generation.scripts.create.target=/usr/local/CreateTables.ddl
```

Example 3, if you want to create the schema using your own (DDL) script, you can set

---

jakarta.persistence.schema-generation.database.action=create jakarta.persistence.schema-generation.create-source=script jakarta.persistence.schema-generation.create-script-source=META-INF/my_create_script.ddl ---

Schema Auto-Generation at runtime

If you want to create the schema (*tables* + *columns* + *constraints*) during the persistence process, the property **datanucleus.schema.autoCreateAll** provides a way of telling DataNucleus to do this. It's a shortcut to setting the other 3 properties to true. Thereafter, during calls to DataNucleus to persist classes or performs queries of persisted data, whenever it encounters a new class to persist that it has no information about, it will use the MetaData to check the datastore for presence of the "table", and if it doesn't exist, will create it. In addition it will validate the correctness of the table (compared to the MetaData for the class), and any other constraints that it requires (to manage any relationships). If any constraints are missing it will create them.

- If you wanted to only create the "tables" required, and none of the "constraints" the property **datanucleus.schema.autoCreateTables** provides this, simply performing the tables part of the above.
- If you want to create any missing "columns" that are required, the property **datanucleus.schema.autoCreateColumns** provides this, validating and adding any missing columns.
- If you wanted to only create the "constraints" required, and none of the "tables" the property **datanucleus.schema.autoCreateConstraints** provides this, simply performing the "constraints" part of the above.
- If you want to keep your schema fixed (i.e don't allow any modifications at runtime) then make

sure that the properties **datanucleus.schema.autoCreate{XXX}** are set to *false*

Schema Generation : Validation

DataNucleus can check any existing schema against what is implied by the MetaData.

The property **datanucleus.schema.validateTables** provides a way of telling DataNucleus to validate any tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the MetaData definition) they would set this property to *true*. This can be useful for example where you are trying to map to an existing schema and want to verify that you've got the correct MetaData definition.

The property **datanucleus.schema.validateColumns** provides a way of telling DataNucleus to validate any columns of the tables that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their tables match what DataNucleus requires (from the MetaData definition) they would set this property to *true*. This will validate the precise column types and widths etc, including defaultability/nullability settings. **Please be aware that many JDBC drivers contain bugs that return incorrect column detail information and so having this turned off is sometimes the only option (dependent on the JDBC driver quality).**

The property **datanucleus.schema.validateConstraints** provides a way of telling DataNucleus to validate any constraints (primary keys, foreign keys, indexes) that it needs against their current definition in the datastore. If the user already has a schema, and want to make sure that their table constraints match what DataNucleus requires (from the MetaData definition) they would set this property to *true*.

Schema Generation : Naming Issues

Some datastores allow access to multiple "schemas" (such as with most RDBMS). DataNucleus will, by default, use the "default" database schema for the Connection URL and user supplied. This may cause issues where the user has been set up and in some databases (e.g Oracle) you want to write to a different schema (which that user has access to). To achieve this in DataNucleus you would set the persistence properties

```
datanucleus.mapping.Catalog={the_catalog_name}
datanucleus.mapping.Schema={the_schema_name}
```

This will mean that all RDBMS DDL and SQL statements will prefix table names with the necessary catalog and schema names (specify which ones your datastore supports).

Some RDBMS do not support specification of both catalog and schema. For example MySQL/MariaDB use catalog and not schema. You need to check what is appropriate for your datastore.

The datastore will define what *case* of identifiers (table/column names) are accepted. By default, DataNucleus will capitalise names (assuming that the datastore supports it). You can however influence the case used for identifiers. This is specifiable with the persistence property **datanucleus.identifier.case**, having the following values

- **UpperCase**: identifiers are in upper case
- **lowercase**: identifiers are in lower case
- **MixedCase**: No case changes are made to the name of the identifier provided by the user (class name or metadata).

Some datastores only support UPPERCASE or lowercase identifiers and so setting this parameter may have no effect if your database doesn't support that option.

This case control only applies to DataNucleus-generated identifiers. If you provide your own identifiers for things like schema/catalog etc then you need to specify those using the case you wish to use in the datastore (including quoting as necessary)

Schema Generation : Column Ordering

By default all tables are generated with columns in alphabetical order, starting with root class fields followed by subclass fields (if present in the same table) etc. This is not part of Jakarta Persistence but DataNucleus allows an extension to specify the relative position, such as

```
@ColumnPosition(3)
```

Note that the values of the position start at 0, and should be specified completely for all columns of all fields.

Schema : Read-Only

If your datastore is read-only (you can't add/update/delete any data in it), obviously you could just configure your application to not perform these operations. An alternative is to set the EMF as read-only, by setting the persistence property **datanucleus.ReadOnlyDatastore** to *true*.

From now on, whenever you perform a persistence operation that implies a change in datastore data, the operation will throw a *PersistenceException*.

DataNucleus provides an additional control over the behaviour when an attempt is made to change a read-only datastore. The default behaviour is to throw an exception. You can change this using the persistence property *datanucleus.readOnlyDatastoreAction* with values of "EXCEPTION" (default), and "IGNORE". "IGNORE" has the effect of simply ignoring all attempted updates to readonly objects.

You can take this read-only control further and specify it just on specific classes. Like this

```
@Extension(vendorName="datanucleus", key="read-only", value="true")
public class MyClass {...}
```

SchemaTool

DataNucleus SchemaTool currently works with RDBMS, HBase, Excel, OOXML, ODF, MongoDB, Cassandra datastores and is very simple to operate. It has the following modes of operation :

- **createDatabase** - create the specified database (catalog/schema) if the datastore supports that operation.
- **deleteDatabase** - delete the specified database (catalog.schema) if the datastore supports that operation.
- **create** - create all database tables required for the classes defined by the input data.
- **delete** - delete all database tables required for the classes defined by the input data.
- **deletecreate** - delete all database tables required for the classes defined by the input data, then create the tables.
- **validate** - validate all database tables required for the classes defined by the input data.
- **dbinfo** - provide detailed information about the database, it's limits and datatypes support. Only for RDBMS currently.
- **schemainfo** - provide detailed information about the database schema. Only for RDBMS currently.

In addition for RDBMS, the **create**/**delete** modes can be used by adding "-ddlFile {filename}" and this will then not create/delete the schema, but instead output the DDL for the tables/constraints into the specified file.

For the **create**, **delete** and **validate** modes DataNucleus SchemaTool accepts either of the following types of input.

- A set of MetaData and class files. The MetaData files define the persistence of the classes they contain. The class files are provided when the classes have annotations.
- The name of a **persistence-unit**. The persistence-unit name defines all classes, metadata files, and jars that make up that unit. Consequently, running DataNucleus SchemaTool with a persistence unit name will create the schema for all classes that are part of that unit.

if using SchemaTool with a persistence-unit make sure you omit *jakarta.persistence.schema-generation* properties from your persistence-unit.

Here we provide many different ways to invoke **DataNucleus SchemaTool**

- Invoke it using Maven, with the DataNucleus Maven plugin

- Invoke it using Ant, using the provided DataNucleus SchemaTool Ant task
- Invoke it manually from the command line
- Invoke it using the DataNucleus Eclipse plugin
- Invoke it programmatically from within an application

SchemaTool using Maven

If you are using Maven to build your system, you will need the DataNucleus Maven plugin. This provides 5 goals representing the different modes of **DataNucleus SchemaTool**. You can use the goals **datanucleus:schema-create**, **datanucleus:schema-delete**, **datanucleus:schema-validate** depending on whether you want to create, delete or validate the database tables. To use the DataNucleus Maven plugin you will may need to set properties for the plugin (in your `pom.xml`). For example

Property	Default	Description
api	JDO	API for the metadata being used (JDO, JPA, Jakarta). **Set this to Jakarta**
ignoreMetaDataForMissingClasses	false	Whether to ignore when we have metadata specified for classes that aren't found
catalogName		Name of the catalog (mandatory when using *createDatabase* or *deleteDatabase* options)
schemaName		Name of the schema (mandatory when using *createDatabase* or *deleteDatabase* options)
persistenceUnitName		Name of the persistence-unit to generate the schema for (defines the classes and the properties defining the datastore). **Mandatory**
log4jConfiguration		Config file location for Log4J (if using it). Note that you can just put a *log4j.properties* at the root of the CLASSPATH instead of specifying this.
log4j2Configuration		Config file location for Log4J v2 (if using it). Note that you can just put a *log4j2.xml* at the root of the CLASSPATH instead of specifying this.
jdkLogConfiguration		Config file location for java.util.logging (if using it)
verbose	false	Verbose output?
fork	true	Whether to fork the SchemaTool process. Note that if you don't fork the process, DataNucleus will likely struggle to determine class names from the input filenames, so you need to use a `persistence.xml` file defining the class names directly.
ddlFile		Name of an output file to dump any DDL to (for RDBMS)
completeDdl	false	Whether to generate DDL including things that already exist? (for RDBMS)

So to give an example, I add the following to my `pom.xml`

```
<build>
    ...
    <plugins>
        <plugin>
            <groupId>org.datanucleus</groupId>
            <artifactId>datanucleus-maven-plugin</artifactId>
            <version>6.0.0-release</version>
            <configuration>
                <api>Jakarta</api>
                <persistenceUnitName>MyUnit</persistenceUnitName>
                <log4jConfiguration>${basedir}/log4j.properties</log4jConfiguration>
                <verbose>true</verbose>
            </configuration>
        </plugin>
    </plugins>
    ...
</build>
```

So with these properties when I run SchemaTool it uses properties from the file `datanucleus.properties` at the root of the Maven project. I am also specifying a log4j (v1) configuration file defining the logging for the SchemaTool process. I then can invoke any of the Maven goals

```
mvn datanucleus:schema-createdatabase      Create the Database (catalog/schema)
mvn datanucleus:schema-deletedatabase      Delete the Database (catalog/schema)
mvn datanucleus:schema-create              Create the tables for the specified classes
mvn datanucleus:schema-delete              Delete the tables for the specified classes
mvn datanucleus:schema-deletecreate        Delete and create the tables for the
specified classes
mvn datanucleus:schema-validate            Validate the tables for the specified
classes
mvn datanucleus:schema-info                Output info for the Schema
mvn datanucleus:schema-dbinfo              Output info for the datastore
```

Schematool using Ant

An Ant task is provided for using **DataNucleus SchemaTool**. It has classname **org.datanucleus.store.schema.SchemaToolTask**, and accepts the following parameters

Parameter	Description	values
api	API that we are using in our use of DataNucleus. **Set this to Jakarta typically**	**JDO**, JPA, Jakarta

Parameter	Description	values
persistenceUnit	Name of the persistence-unit that we should manage the schema for (defines the classes and the properties defining the datastore).	
mode	Mode of operation.	**create**, delete, validate, dbinfo, schemainfo, createDatabase, deleteDatabase
catalogName	Catalog name to use when used in *createDatabase* /*deleteDatabase* modes	
schemaName	Schema name to use when used in *createDatabase* /*deleteDatabase* modes	
verbose	Whether to give verbose output.	true, **false**
ddlFile	The filename where SchemaTool should output the DDL (for RDBMS).	
completeDdl	Whether to output complete DDL (instead of just missing tables). Only used with ddlFile	true, **false**

The SchemaTool task extends the Apache Ant Java task, thus all parameters available to the Java task are also available to the SchemaTool task.

In addition to the parameters that the Ant task accepts, you will need to set up your CLASSPATH to include the classes and MetaData files, and to define the following system properties via the *sysproperty* parameter (not required when specifying the persistence props via the properties file, or when providing the *persistence-unit*)

Parameter	Description	Optional
datanucleus.ConnectionURL	URL for the database	✔
datanucleus.ConnectionUserName	User name for the database	✔
datanucleus.ConnectionPassword	Password for the database	✔
datanucleus.ConnectionDriverName	Name of JDBC driver class	✔
log4j.configuration	Log4J (v1) configuration file, for SchemaTool's Log	✘
log4j.configurationFile	Log4J (v2) configuration file, for SchemaTool's Log	✘

So you could define something *like* the following, setting up the parameters **schematool.classpath**, **datanucleus.ConnectionURL**, **datanucleus.ConnectionUserName**, **datanucleus.ConnectionPassword**(, **datanucleus.ConnectionDriverName**) to suit your situation.

Schematool Command-Line Usage

If you wish to call **DataNucleus SchemaTool** manually, it can be called as follows

```
java [-cp classpath] [system_props] org.datanucleus.store.schema.SchemaTool [modes]
[options]
    where system_props (when specified) should include
        -Ddatanucleus.ConnectionURL=db_url
        -Ddatanucleus.ConnectionUserName=db_username
        -Ddatanucleus.ConnectionPassword=db_password
        -Dlog4j.configuration=file:{log4j.properties} (optional)
        -Dlog4j.configurationFile=file:{log4j2.xml} (optional)
    where modes can be
        -createDatabase : create the specified database (if supported)
        -deleteDatabase : delete the specified database (if supported)
        -create : Create the tables specified by the mapping-files/class-files
        -delete : Delete the tables specified by the mapping-files/class-files
        -deletecreate : Delete the tables specified by the mapping-files/class-files
and then create them
        -validate : Validate the tables specified by the mapping-files/class-files
        -dbinfo : Detailed information about the database
        -schemainfo : Detailed information about the database schema
    where options can be
        -catalog {catalogName} : Catalog name when using
"createDatabase"/"deleteDatabase"
        -schema {schemaName} : Schema name when using
"createDatabase"/"deleteDatabase"
        -api : The API that is being used (default is JDO, but set this to Jakarta)
        -pu {persistence-unit-name} : Name of the persistence unit to manage the
schema for
        -ddlFile {filename} : RDBMS - only for use with "create"/"delete" mode to dump
the DDL to the specified file
        -completeDdl : RDBMS - when using "ddlFile" in "create" mode to get all DDL
output and not just missing tables/constraints
        -v : verbose output
```

All classes, MetaData files, `persistence.xml` files must be present in the CLASSPATH. In terms of the schema to use, you either specify the "props" file (recommended), or you specify the System properties defining the database connection, or the properties in the "persistence-unit". You should only specify one of the modes above. Let's make a specific example and see the output from SchemaTool. So we have the following files in our application

```
src/java/...                (source files and MetaData files)
target/classes/...          (enhanced classes, and MetaData files)
lib/log4j.jar               (optional, for Log4J logging)
lib/datanucleus-core.jar
lib/datanucleus-api-jakarta.jar
lib/datanucleus-rdbms.jar, lib/datanucleus-hbase.jar,  etc
lib/jakarta.persistence.jar
lib/mysql-connector-java.jar (driver for our database)
log4j.properties
```

So we want to create the schema for our persistent classes. So let's invoke **DataNucleus SchemaTool** to do this, from the top level of our project. In this example we're using Linux (change the CLASSPATH definition to suit for Windows)

```
java -cp target/classes:lib/log4j.jar:lib/datanucleus-core.jar:lib/datanucleus-
{datastore}.jar:lib/mysql-connector-java.jar
      -Dlog4j.configuration=file:log4j.properties
      org.datanucleus.store.schema.SchemaTool -create
      -api Jakarta -pu MyUnit

DataNucleus SchemaTool (version 5.0.0.release) : Creation of the schema

DataNucleus SchemaTool : Classpath
>>  /home/andy/work/DataNucleus/samples/packofcards/target/classes
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/log4j.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-core.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-api-jakarta.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/jakarta.persistence.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-rdbms.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/mysql-connector-java.jar

DataNucleus SchemaTool : Persistence-Unit="MyUnit"

SchemaTool completed successfully
```

So as you see, **DataNucleus SchemaTool** prints out our input, the properties used, and finally a success message. If an error occurs, then something will be printed to the screen, and more information will be written to the log.

SchemaTool API

DataNucleus SchemaTool can also be called programmatically from an application. You need to get hold of the StoreManager and cast it to *SchemaAwareStoreManager*. The API is shown below.

```
package org.datanucleus.store.schema;

public interface SchemaAwareStoreManager
{
    void createDatabase(String catalogName, String schemaName, Properties props);
    void deleteDatabase(String catalogName, String schemaName, Properties props);

    void createSchemaForClasses(Set<String> classNames, Properties props);
    void deleteSchemaForClasses(Set<String> classNames, Properties props);
    void validateSchemaForClasses(Set<String> classNames, Properties props);
}
```

So for example to create the schema for classes *mydomain.A* and *mydomain.B* you would do something like this

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MyUnit");
PersistenceNucleusContext nucCtx = emf.unwrap(PersistenceNucleusContext.class);
...
List classNames = new ArrayList();
classNames.add("mydomain.A");
classNames.add("mydomain.B");
try
{
    Properties props = new Properties();
    // Set any properties for schema generation
    ((SchemaAwareStoreManager)nucCtx.getStoreManager()).createSchemaForClasses
(classNames, props);
}
catch(Exception e)
{
    ...
}
```

Schema Adaption

As time goes by during the development of your DataNucleus Jakarta powered application you may need to add fields, update field mappings, or delete fields. In an ideal world the Jakarta provider would take care of this itself. However this is actually not part of the Jakarta Persistence standard and so you are reliant on what features the Jakarta provider possesses.

DataNucleus can cope with added fields, if you have the relevant persistence properties enabled. In this case look at **datanucleus.schema.autoCreateTables**, **datanucleus.schema.autoCreateColumns**, **datanucleus.schema.autoCreateConstraints**, and **datanucleus.rdbms.dynamicSchemaUpdates** (with this latter property of use where you have interface field(s) and a new implementation of that interface is encountered at runtime).

If you **update** or **delete** a field with an RDBMS datastore then you will need to update your schema

manually. With non-RDBMS datastores deletion of fields is supported in some situations.

You should also consider making use of tools like Flyway and Liquibase since these are designed for exactly this role.

RDBMS : Datastore Schema SPI

The JPA API doesn't provide a way of accessing the schema of the datastore itself (if it has one). In the case of RDBMS it is useful to be able to find out what columns there are in a table, or what data types are supported for example. DataNucleus Access Platform provides an API for this.

The first thing to do is get your hands on the DataNucleus *StoreManager* and from that the *StoreSchemaHandler*. You do this as follows

```
import org.datanucleus.store.StoreManager;
import org.datanucleus.store.schema.StoreSchemaHandler;

...
StoreManager storeMgr = emf.unwrap(StoreManager.class);
StoreSchemaHandler schemaHandler = storeMgr.getSchemaHandler();
```

So now we have the *StoreSchemaHandler* what can we do with it? Well start with the javadoc for the implementation that is used for RDBMS Javadoc

RDBMS : Datastore Types Information

So we now want to find out what JDBC/SQL types are supported for our RDBMS. This is simple.

```
import org.datanucleus.store.rdbms.schema.RDBMSTypesInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTypesInfo typesInfo = schemaHandler.getSchemaData(conn, "types");
```

As you can see from the javadocs for *RDBMSTypesInfo* Javadoc we can access the JDBC types information via the "children". They are keyed by the JDBC type number of the JDBC type (see java.sql.Types). So we can just iterate it

```
Iterator jdbcTypesIter = typesInfo.getChildren().values().iterator();
while (jdbcTypesIter.hasNext())
{
    JDBCTypeInfo jdbcType = (JDBCTypeInfo)jdbcTypesIter.next();

    // Each JDBCTypeInfo contains SQLTypeInfo as its children, keyed by SQL name
    Iterator sqlTypesIter = jdbcType.getChildren().values().iterator();
    while (sqlTypesIter.hasNext())
    {
        SQLTypeInfo sqlType = (SQLTypeInfo)sqlTypesIter.next();
        ... inspect the SQL type info
    }
}
```

RDBMS : Column information for a table

Here we have a table in the datastore and want to find the columns present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableInfo tableInfo = schemaHandler.getSchemaData(conn, "columns",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableInfo* Javadoc we can access the columns information via the "children".

```
Iterator columnsIter = tableInfo.getChildren().iterator();
while (columnsIter.hasNext())
{
    RDBMSColumnInfo colInfo = (RDBMSColumnInfo)columnsIter.next();

    ...
}
```

RDBMS : Index information for a table

Here we have a table in the datastore and want to find the indices present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableIndexInfo tableInfo = schemaHandler.getSchemaData(conn, "indices",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableIndexInfo* Javadoc we can access the index information via the "children".

```
Iterator indexIter = tableInfo.getChildren().iterator();
while (indexIter.hasNext())
{
    IndexInfo idxInfo = (IndexInfo)indexIter.next();

    ...
}
```

RDBMS : ForeignKey information for a table

Here we have a table in the datastore and want to find the FKs present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTableFKInfo tableInfo = schemaHandler.getSchemaData(conn, "foreign-keys",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTableFKInfo* Javadoc we can access the foreign-key information via the "children".

```
Iterator fkIter = tableInfo.getChildren().iterator();
while (fkIter.hasNext())
{
    ForeignKeyInfo fkInfo = (ForeignKeyInfo)fkIter.next();

    ...
}
```

RDBMS : PrimaryKey information for a table

Here we have a table in the datastore and want to find the PK present. So we do this

```
import org.datanucleus.store.rdbms.schema.RDBMSTableInfo;

Connection conn = (Connection)pm.getDataStoreConnection().getNativeConnection();
RDBMSTablePKInfo tableInfo = schemaHandler.getSchemaData(conn, "primary-keys",
    new Object[] {catalogName, schemaName, tableName});
```

As you can see from the javadocs for *RDBMSTablePKInfo* Javadoc we can access the foreign-key information via the "children".

```
Iterator pkIter = tableInfo.getChildren().iterator();
while (pkIter.hasNext())
{
    PrimaryKeyInfo pkInfo = (PrimaryKeyInfo)pkIter.next();

    ...
}
```

EntityManager

Now that we have our EntityManagerFactory, providing the connection for our *persistence-unit* to our datastore, we need to obtain an *EntityManager* (EM) to manage the persistence of objects. Here we describe the majority of operations that you will are likely to need to know about.

An *EntityManagerFactory* is designed to be thread-safe. An *EntityManager* is not. If you set the persistence property **datanucleus.Multithreaded** this acts as a hint to the EMF to provide *EntityManager(s)* that are usable with multiple threads. While DataNucleus makes efforts to make this *EntityManager* usable with multiple threads, it is not guaranteed to work multi-threaded in all situations, particularly around second class collection/map fields.

An *EntityManager* is cheap to create and it is a common pattern for web applications to open an *EntityManager* per web request, and close it before the response. Always close your *EntityManager* after you have finished with it.

To take an example, suppose we have the following (abbreviated) entities

```
@Entity
public class Person
{
    @Id
    long id;

    String firstName;
    String lastName;
}

@Entity
public class Account
{
    @Id
    long id;

    @OneToOne
    Person person;
}
```

Opening/Closing an EntityManager

You obtain an *EntityManager* Javadoc from the *EntityManagerFactory* as follows

```
EntityManager em = emf.createEntityManager();
```

In the case of using container-managed JavaEE, you would instead obtain the *EntityManager* by

injection

```
@PersistenceContext(unitName="myPU")
EntityManager em;
```

You then perform all operations that you need using this *EntityManager*.

If you manually created the *EntityManager* using *createEntityManager()* then you must also **close** it; forgetting to close it will lead to memory/resource leaks.

```
em.close();
```

In general you will be performing all operations on a *EntityManager* within a transaction, whether your transactions are controlled by your JavaEE container, by a framework such as Spring, or by locally defined transactions. In the examples below we will omit the transaction demarcation for clarity.

Persisting an Object

The main thing that you will want to do with the data layer of a Jakarta-enabled application is persist your objects into the datastore. As we mentioned earlier, a *EntityManagerFactory* represents the datastore where the objects will be persisted. So you create a normal Java object in your application, and you then persist this as follows

```
Person lincoln = new Person(1, "Abraham", "Lincoln");
em.persist(lincoln);
```

This will result in the object being persisted into the datastore, though clearly it will not be persistent until you commit the transaction. The Lifecycle State of the object changes from *Transient* to *Persistent* (after persist()), to *Persistent*/*Detached* (at commit).

Persisting multiple Objects in one call

When you want to persist multiple objects with standard Jakarta Persistence you have to call *persist* multiple times. Fortunately DataNucleus extends this to take in a Collection or an array of entities, so you can do

```
Collection<Person> coll = new HashSet<>();
coll.add(lincoln);
coll.add(mandela);

em.persist(coll);
```

As above, the objects are persisted to the datastore. The Lifecycle State of the objects change from *Transient* to *Persistent* (after persist()), to *Persistent*/*Detached* (at commit).

Finding an object by its identity

Once you have persisted an object, it has an "identity". This is a unique way of identifying it. When you specify the persistence for the entity you specified an id field (or fields, together with an *IdClass*) so you can create the identity from that. So what ? Well the identity can be used to retrieve the object again at some other part in your application. So you pass the identity into your application, and the user clicks on some button on a web page and that button corresponds to a particular object identity. You can then go back to your data layer and retrieve the object as follows

```
Person p = em.find(Person.class, 1);
```

which will try to retrieve the *Person* object with identity of 1. If there is no *Person* object with that identity then it returns *null*.

the first argument could be a base class and the real object could be an instance of a subclass of that.

the second argument is either the value of the single primary-key field (when it has only one `@Id` field), or is the value of the *object-id-class* (when it has multiple `@Id` fields).

if the second argument is not of the type expected for the `@Id` field then it will throw an exception. You can enable DataNucleus built-in type conversion by setting the persistence property **datanucleus.findObject.typeConversion** to *true*.

Finding an object by its class and unique key field value(s)

Whilst the primary way of looking up an object is via its *identity*, in some cases a class has a *unique key* (maybe comprised of multiple field values). This is sometimes referred to as a *natural id*. This is not part of the Jakarta Persistence API, however DataNucleus makes it available. Let's take an example

```
@Entity
@Table(uniqueConstraints={@UniqueConstraint(columnNames={"firstName","lastName"})})
public class Person
{
    @Id
    long id;

    LocalDate dob;

    String firstName;

    String lastName;

    int age;

    ...
}
```

Here we have a *Person* class with an identity defined as a long, but also with a *unique key* defined as the composite of the *firstName* and *lastName* (in most societies it is possible to duplicate names amongst people, but we just take this as an example).

Now to access a *Person* object based on the *firstName* and *lastName* we do the following

```
JakartaEntityManager jakartaEM = (JakartaEntityManager)em;
Person p = jakartaEM.findByUnique(Person.class, {"firstName", "lastName"}, {"George",
"Jones"});
```

and we retrieve the *Person* "George Jones".

Deleting an Object

When you need to delete an object that you had previous persisted, deleting it is simple. Firstly you need to get the object itself, and then delete it as follows

```
Person lincoln = em.find(Person.class, 1);  // Retrieves the object to delete
em.remove(lincoln);
```

Deleting multiple Objects

When you want to delete multiple objects with standard Jakarta Persistence you have to call *remove* multiple times. Fortunately DataNucleus extends this to take in a Collection or an array of entities, so you can do

```
Collection<Person> people = new HashSet<>();
people.add(lincoln);
people.add(mandela);
em.remove(people);
```

Modifying a persisted Object

To modify a previously persisted object you take the object and update it in your code. If the object is in "detached" state (not managed by a particular *EntityManager*) then when you are ready to persist the changes you do the following

```
Object updatedObj = em.merge(obj);
```

If however the object was already managed at the point of updating its fields, then

```
Person lincoln = em.find(Person.class, 1); // "lincoln" is now managed by "em", and in
"persistent" state.

lincoln.setAddress("The White House");
```

when the *setAddress* has been called, this is intercepted by DataNucleus, and the changes will be stored for persisting. There is no need to call any *EntityManager* method to push the changes. This is part of the mechanism known as *transparent persistence*.

Modifying multiple persisted Objects

When you want to attach multiple modified objects with standard Jakarta Persistence you have to call *merge* multiple times. Fortunately DataNucleus extends this to take in a Collection or an array of entities, so you can do

```
Object updatedObj = em.merge(coll);
```

Refreshing a persisted Object

An application that has sole access to the datastore, in general, does not need to check for updated values from the datastore. In more complicated situations the datastore may be updated by another application for example, so it may be necessary at times to check for more up-to-date values for the fields of an entity. You do that like this

```
em.refresh(lincoln);
```

This will do the following

- Refresh all fields that are to be eagerly fetched from the datastore
- Unload all loaded fields that are to be lazily fetched.

If the object had any changes they will be thrown away by this step, and replaced by the latest datastore values.

Getting EntityManager for an object

Jakarta Persistence doesn't provide a method for getting the EntityManager of an object as such. Fortunately DataNucleus provides the following

```
import org.datanucleus.api.jakarta.NucleusJakartaHelper;

...

EntityManager em = NucleusJakartaHelper.getEntityManager(obj);
```

If you have an *EntityManager* object and want to check if it is managing a particular object you can call

```
boolean managedByThisEM = em.contains(lincoln);
```

Cascading Operations

When you have relationships between entities, and you persist one entity, by default the related entity will *not* be persisted. For each of the relation annotations `@OneToOne`, `@OneToMany`, `@ManyToOne` and `@ManyToMany` there is an attribute *cascade* which defaults to null but you can control what operations cascade (persist, remove, merge, detach, refresh).

Let's use our example above, and create new *Person* and *Account* objects.

```
Person lincoln = new Person(1, "Abraham", "Lincoln");
Account acct1 = new Account(1, lincoln); // Second argument sets the relation between
the objects
```

now to persist them both we have two options. Firstly with the default cascade setting

```
em.persist(lincoln);
em.persist(acct1);
```

The second option is to set the metadata on *Account* as

```
@Entity
public class Account
{
    @Id
    long id;

    @OneToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE})
    Person person;
}
```

now we can simply do this (since the *Account* has a reference to *Person*);

```
em.persist(acct1);
```

Orphans

When an element is removed from a collection, or when a 1-1 relation is nulled, sometimes it is desirable to delete the other object. Jakarta Persistence defines a facility of removing "orphans" by specifying metadata for a 1-1 or 1-N relation. Let's take our example. In the above relation between *Account* and *Person* if we set the "person" field to null, this should mean that the *Person* record is deleted. So we could change the metadata to

```
@Entity
public class Account
{
    @Id
    long id;

    @OneToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE}, `orphanRemoval`=true)
    Person person;
}
```

So from now on, if we delete the *Account* we delete the *Person*, and if we set the "person" field of *Account* to null then we also delete the *Person*.

Managing Relationships

The power of a Java persistence solution like DataNucleus is demonstrated when persisting relationships between objects. There are many types of relationships.

- 1-1 relationships - this is where you have an object A relates to a second object B. The relation can be *unidirectional* where A knows about B, but B doesnt know about A. The relation can be *bidirectional* where A knows about B and B knows about A.
- 1-N relationships - this is where you have an object A that has a collection of other objects of type B. The relation can be *unidirectional* where A knows about the objects B but the Bs dont know about A. The relation can be *bidirectional* where A knows about the objects B and the Bs know about A
- N-1 relationships - this is where you have an object B1 that relates to an object A, and an object B2 that relates to A also etc. The relation can be *unidirectional* where the A doesnt know about the Bs. The relation can be *bidirectional* where the A has a collection of the Bs. i.e a 1-N relationship but from the point of view of the element.
- M-N relationships - this is where you have objects of type A that have a collection of objects of type B and the objects of type B also have a collection of objects of type A. The relation is always *bidirectional* by definition
- Derived Identity relationships when you have a relation and part of the primary key of the related object is the other persistent object.

Assigning Relationships

When the relation is *unidirectional* you simply set the related field to refer to the other object. For example we have classes A and B and the class A has a field of type B. So we set it like this

```
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
```

When the relation is *bidirectional* you **have to set both sides** of the relation. For example, we have classes A and B and the class A has a collection of elements of type B, and B has a field of type A. So we set it like this

```
A a = new A();
B b1 = new B();
a.addElement(b1); // "a" knows about "b1"
b1.setA(a); // "b1" knows about "a"
```

With a *bidirectional* relation you must set both sides of the relation

Persisting Relationships - Reachability

To persist an object with Jakarta Persistence you call the *EntityManager* method *persist* (or *merge* if wanting to update a detached object). The object passed in will be persisted. By default all related objects will **not** be persisted with that object. You can however change this by specifying the *cascade* PERSIST (and/or MERGE) property for that field. With this the related object(s) would also be persisted (or updated with any new values if they are already persistent). This process is called

persistence-by-reachability. For example we have classes A and B and class A has a field of type B and this field has the *cascade* property PERSIST set. To persist them we could do

```
A a = new A();
B b = new B();
a.setB(b);
em.persist(a); // "a" and "b" are provisionally persistent
```

A further example where you don't have the *cascade* PERSIST set, but still want to persist both ends of a relation.

```
A a = new A();
B b = new B();
a.setB(b);
em.persist(a); // "a" is provisionally persistent
em.persist(b); // "b" is provisionally persistent
```

Managed Relationships

As we have mentioned above, it is for the user to set both sides of a bidirectional relation. If they don't and object A knows about B, but B doesnt know about A then what is the persistence solution to do ? It doesn't know which side of the relation is correct. Jakarta Persistence doesn't define the behaviour for this situation. DataNucleus has two ways of handling this situation. If you have the persistence property **datanucleus.manageRelationships** set to true then it will make sure that the other side of the relation is set correctly, correcting obvious omissions, and giving exceptions for obvious errors. If you set that persistence property to false then it will assume that your objects have their bidirectional relationships consistent and will just persist what it finds.

When performing management of relations there are some checks implemented to spot typical errors in user operations e.g add an element to a collection and then remove it (why?!). You can disable these checks using **datanucleus.manageRelationshipsChecks**, set to false.

Level 1 Cache

Each EntityManager maintains a cache of the objects that it has encountered (or have been "enlisted") during its lifetime. This is termed the **Level 1 (L1) Cache**. It is enabled by default and you should only ever disable it if you really know what you are doing. There are inbuilt types for the L1 Cache available for selection. DataNucleus supports the following types of L1 Cache :-

- *weak* - uses a weak reference backing map. If JVM garbage collection clears the reference, then the object is removed from the cache.
- *soft* - uses a soft reference backing map. If the map entry value object is not being actively used, then garbage collection *may* garbage collect the reference, in which case the object is removed from the cache.

- *strong* - uses a normal HashMap backing. With this option all references are strong meaning that objects stay in the cache until they are explicitly removed by calling remove() on the cache.

You can specify the type of L1 Cache by providing the persistence property **datanucleus.cache.level1.type**. You set this to the value of the type required. If you want to remove all objects from the L1 cache programmatically you should use *em.clear()* but bear in mind the other things that this will impact on.

Objects are placed in the L1 Cache (and updated there) during the course of the transaction. This provides rapid access to the objects in use in the users application and is used to guarantee that there is only one object with a particular identity at any one time for that EntityManager. When the EntityManager is closed the cache is cleared.

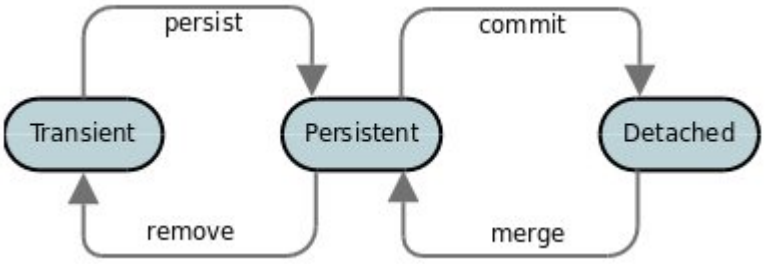
The L1 cache is a DataNucleus 

allowing you to provide your own cache where you require it.

Object Lifecycle

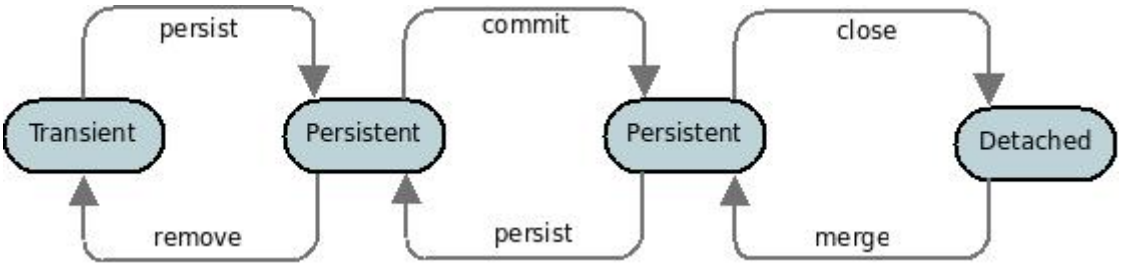
During the persistence process an object goes through lifecycle changes. Below we demonstrate the primary object lifecycle changes for Jakarta Persistence. With Jakarta Persistence these lifecycles are referred to as "persistence contexts". There are two : *transaction* (default for JavaEE usage) and *extended* (default for JavaSE usage). DataNucleus allows control over which to use by specification of the persistence property **datanucleus.jakarta.persistenceContextType**.

Transaction PersistenceContext



A newly created object is **transient**. You then persist it and it becomes **persistent**. You then commit the transaction and it is detached for use elsewhere in the application, in **detached** state. You then attach any changes back to persistence and it becomes **persistent** again. Finally when you delete the object from persistence and commit that transaction it is in **transient** state.

Extended PersistenceContext



So a newly created object is **transient**. You then persist it and it becomes **persistent**. You then commit the transaction and it remains managed in **persistent** state. When you close the EntityManager it becomes **detached**. Finally when you delete the object from persistence and commit that transaction it is in **transient** state.

Detachment

When you detach an object (and its graph) either explicitly (using em.detach()) or implicitly via the PersistenceContext above, you need to be careful about which fields are detached. If you detach everything then you can end up with a huge graph that could impact on the performance of your application. On the other hand you need to ensure that you have all fields that you will be needing access to whilst detached. Should you access a field that was not detached an **IllegalAccessException** is thrown. All fields that are loaded will be detached so make sure you either load all required when retrieving the object using Entity Graphs or you access fields whilst attached (which will load them).

Please note that some people interpret the Jakarta Persistence spec as implying that an object which has a primary key field set to a value as being *detached*. DataNucleus does **not** take this point of view, since the only way you can have a detached object is to detach it from persistence (i.e it was once managed/attached). To reinforce our view of things, what state is an object in which has a primitive primary key field ? Using the logic above of these other people any object of such a class would be in *detached* state (when not managed) since its PK is set. **An object that has a PK field set is *transient* unless it was detached from persistence**. Note that you can *merge* a transient object by setting the persistence property **datanucleus.allowAttachOfTransient** to *true*.

DataNucleus does not use the "CascadeType.DETACH" flag explicitly, and instead detaches the fields that are loaded (or marked for eager loading). In addition it allows the user to make use of the *FetchPlan* extension for controlling the fine details of what is loaded (and hence detached).

Helper Methods

Jakarta Persistence provides nothing to determine the lifecycle state of an object. Fortunately DataNucleus does consider this useful, so you can call the following

```
String state = NucleusJakartaHelper.getObjectState(entity);
boolean detached = NucleusJakartaHelper.isDetached(entity);
boolean persistent = NucleusJakartaHelper.isPersistent(entity);
boolean deleted = NucleusJakartaHelper.isDeleted(entity);
boolean transactional = NucleusJakartaHelper.isTransactional(entity);
```

When an object is detached it is often useful to know which fields are loaded/dirty. You can do this with the following helper methods

```
Object[] detachedState = NucleusJakartaHelper.getDetachedStateForObject(entity);
// detachedState[0] is the identity, detachedState[1] is the version when detached
// detachedState[2] is a BitSet for loaded fields
// detachedState[3] is a BitSet for dirty fields

String[] dirtyFieldNames = NucleusJakartaHelper.getDirtyFields(entity, em);

String[] loadedFieldNames = NucleusJakartaHelper.getLoadedFields(entity, em);
```

Transactions

Persistence operations performed by the *EntityManager* are typically managed in a *transaction*, allowing operations to be grouped together. A Transaction forms a unit of work. The Transaction manages what happens within that unit of work, and when an error occurs the Transaction can roll back any changes performed. Transactions can be managed by the users application, or can be managed by a framework (such as Spring), or can be managed by a JavaEE container. These are described below.

- Local transactions : managed using the Jakarta Transaction API
- JTA transactions : managed using the JTA UserTransaction API
- Container-managed transactions : managed by a JavaEE environment
- Spring-managed transactions : managed by SpringFramework
- No transactions : "auto-commit" mode
- Controlling transaction isolation level
- Read-Only transactions
- Flushing a Transaction
- RDBMS : Savepoints

Locally-Managed Transactions

If using DataNucleus Jakarta in a JavaSE environment the normal type of transaction is *RESOURCE_LOCAL*. With this type of transaction the user manages the transactions themselves, starting, committing or rolling back the transaction. With these transactions with Jakarta Persistence you obtain an *EntityTransaction* Javadoc from the *EntityManager*, and manage it like this

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
try
{
    tx.begin();

    {users code to persist objects}

    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
em.close();
```

In this case you will have defined your persistence-unit to be like this

```
<persistence-unit name="MyUnit" transaction-type="RESOURCE_LOCAL">
    <properties>
        <property key="jakarta.persistence.jdbc.url" value="jdbc:mysql:..."/>
        ...
    </properties>
    ...
</persistence-unit>
```

or

```
<persistence-unit name="MyUnit" transaction-type="RESOURCE_LOCAL">
    <non-jta-data-source>java:comp/env/myDS</properties>
    ...
</persistence-unit>
```

The basic idea with **Locally-Managed transactions** is that you are managing the transaction start and end.

JTA Transactions

Strict Jakarta Persistence does not support use of JTA transactions in a JavaSE environment. DataNucleus does however allow JTA transactions in a JavaSE environment.

The other type of transaction with Jakarta Persistence is using JTA. With this type, where you have a JTA data source from which you have a *UserTransaction*. This *UserTransaction* can have resources "joined" to it. In the case of Jakarta Persistence, you have two scenarios. The first scenario is where you have the *UserTransaction* created before you create your *EntityManager*. The create of the *EntityManager* will automatically join it to the current *UserTransaction*, like this

```
UserTransaction ut = (UserTransaction)new InitialContext().lookup
("java:comp/UserTransaction");
ut.setTransactionTimeout(300);

EntityManager em = emf.createEntityManager();
try
{
    ut.begin();

    .. perform persistence/query operations

    ut.commit();
}
finally
{
    em.close();
}
```

so we control the transaction using the *UserTransaction*.

The second scenario is where the *UserTransaction* is started after you have the *EntityManager*. In this case we need to join our *EntityManager* to the newly created *UserTransaction*, like this

```
EntityManager em = emf.createEntityManager();
try
{
    .. perform persistence, query operations

    UserTransaction ut = (UserTransaction)new InitialContext().lookup
("java:comp/UserTransaction");
    ut.setTransactionTimeout(300);
    ut.begin();

    // Join the EntityManager operations to this UserTransaction
    em.joinTransaction();

    // Commit the persistence/query operations performed above
    ut.commit();
}
finally
{
    em.close();
}
```

In the JTA case you will have defined your persistence-unit to be like this

```
<persistence-unit name="MyUnit" transaction-type="JTA">
    <jta-data-source>java:comp/env/myDS</properties>
    ...
</persistence-unit>
```

JTA TransactionManager

Note that the JavaEE spec does not define a standard way of finding the JTA TransactionManager, and so all JavaEE containers have their own ways of handling this. DataNucleus provides a way of scanning the various methods to find that appropriate for the JavaEE container in use, but you can explicitly set the method of finding the *TransactionManager*, by use of the persistence properties **datanucleus.transaction.jta.transactionManagerLocator** and, if using this property set to *custom_jndi* then also **datanucleus.transaction.jta.transactionManagerJNDI** set to the JNDI location that stores the *TransactionManager* instance.

Container-Managed Transactions

When using a JavaEE container you are giving over control of the transactions to the container. Here you have **Container-Managed Transactions**. In terms of your code, you would do like the above examples **except** that you would OMIT the *tx.begin(), tx.commit(), tx.rollback()* since the JavaEE container will be doing this for you.

Spring-Managed Transactions

When you use a framework like Spring you would not need to specify the *tx.begin(), tx.commit(), tx.rollback()* since that would be done for you.

No Transactions

DataNucleus allows the ability to operate without transactions. With Jakarta Persistence this is enabled by default (see the 2 properties **datanucleus.transaction.nontx.read**, **datanucleus.transaction.nontx.write** set to *true*, the default). This means that you can read objects and make updates outside of transactions. This is effectively an "auto-commit" mode.

```
EntityManager em = emf.createEntityManager();

{users code to persist objects}

em.close();
```

When using non-transactional operations, you need to pay attention to the persistence property **datanucleus.transaction.nontx.atomic**. If this is true then any persist/delete/update will be committed to the datastore immediately. If this is false then any persist/delete/update will be queued up until the next transaction (or *em.close()*) and committed with that.

Some other Jakarta providers do not provide this flexibility of non-transactional handling, and indeed, if you try to do updates when outside a transaction these changes are not committed even at *em.close* with those Jakarta providers. Fortunately you're using DataNucleus and it doesn't have that problem.

Transaction Isolation

DataNucleus also allows specification of the transaction isolation level, applied at the connection level, and providing a level of isolation of this process from other processed using the same database. The isolation is specified via the persistence property **datanucleus.transaction.isolation**. It accepts the standard JDBC values of

- **read-uncommitted** (1) : dirty reads, non-repeatable reads and phantom reads can occur
- **read-committed** (2) : dirty reads are prevented; non-repeatable reads and phantom reads can occur. **This is the default**
- **repeatable-read** (4) : dirty reads and non-repeatable reads are prevented; phantom reads can occur
- **serializable** (8) : dirty reads, non-repeatable reads and phantom reads are prevented

If the datastore doesn't support a particular isolation level then it will silently be changed to one that is supported. As an alternative you can also specify it on a per-transaction basis as follows

```
org.datanucleus.api.jakarta.JakartaEntityTransaction tx = (org.datanucleus.api.
jakarta.JakartaEntityTransaction)em.getTransaction();
tx.setOption("transaction.isolation", "read-committed");
```

Alternatively with numeric input (using numbers in parentheses above).

```
org.datanucleus.api.jakarta.JakartaEntityTransaction tx = (org.datanucleus.api.
jakarta.JakartaEntityTransaction)em.getTransaction();
tx.setOption("transaction.isolation", 2);
```

Read-Only Transactions

Obviously transactions are intended for committing changes. If you come across a situation where you don't want to commit anything under any circumstances you can mark the transaction as "read-only" by calling

```
EntityManager em = emf.createEntityManager();
Transaction tx = em.getTransaction();
try
{
    tx.begin();
    tx.setRollbackOnly();

    {users code to persist objects}

    tx.rollback();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
em.close();
```

Any call to *commit* on the transaction will throw an exception forcing the user to roll it back.

Flushing

During a transaction, depending on the configuration, operations don't necessarily go to the datastore immediately, often waiting until *commit*. In some situations you need persists/updates/deletes to be in the datastore so that subsequent operations can be performed that rely on those being handled first. In this case you can **flush** all outstanding changes to the datastore using

```
em.flush();
```

You can control the *flush mode* for an EntityManager using

```
em.setFlushMode(FlushModeType.COMMIT);
```

which will only flush changes at commit. This means that when a query is performed it will not see any local changes.

The default is FlushModeType.AUTO which will flush just before any query, so that the results of all queries are consistent with local changes.

A convenient vendor extension is to find which objects are waiting to be flushed at any time, like this

```
List<DNStateManager> objs = em.unwrap(ExecutionContext.class).getObjectsToBeFlushed();
```

Transactions with lots of data

Occasionally you may need to persist large amounts of data in a single transaction. Since all objects need to be present in Java memory at the same time, you can get *OutOfMemory* errors, or your application can slow down as swapping occurs. You can alleviate this by changing how you flush/commit the persistent changes.

One way is to do it like this, where possible,

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
try
{
    tx.begin();
    for (int i=0; i<100000; i++)
    {
        Wardrobe wardrobe = new Wardrobe();
        wardrobe.setModel("3 doors");
        em.persist(wardrobe);
        if (i % 10000 == 0)
        {
            // Flush every 10000 objects
            em.flush();
        }
    }
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    em.close();
}
```

Another way, if one object is causing the persist of a huge number of related objects, is to just persist some objects without relations first, flush, and then form the relations. This then allows the above process to be utilised, manually flushing at intervals.

You can additionally consider evicting objects from the Level 1 Cache, since they will, by default, be cached until commit.

Transaction Savepoints

Applicable to RDBMS

JDBC provides the ability to specify a point in a transaction and rollback to that point if required, assuming the JDBC driver supports it. DataNucleus provides this as a vendor extension, as follows

```
import org.datanucleus.api.jakarta.JakartaEntityTransaction;

EntityManager em = emf.createEntityManager();
JakartaEntityTransaction tx = (JakartaEntityTransaction)em.getTransaction();
try
{
    tx.begin();

    {users code to persist objects}
    tx.setSavepoint("Point1");

    {more user code to persist objects}
    tx.rollbackToSavepoint("Point1");

    tx.releaseSavepoint("Point1");
    tx.rollback();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
em.close();
```

Locking

Within a transaction it is very common to require some form of locking of objects so that you can guarantee the integrity of data that is committed. There are the following locking types for a transaction.

- Assume that things in the datastore will not change until they are ready to commit, not lock any records and then just before committing make a check for changes. This is known as Optimistic Locking.
- Lock specific records in a datastore and keep them locked until commit of the changes. These are known as Pessimistic (or datastore) Locking.

Optimistic Locking

Optimistic "locking" is suitable for longer lived operations maybe where user interaction is taking place and where it would be undesirable to block access to datastore entities for the duration of the transaction. The assumption is that data altered in this transaction will not be updated by other transactions during the duration of this transaction, so the changes are not propagated to the datastore until commit()/flush(). The obvious benefit of optimistic locking is that all changes are made in a block and version checking of objects is performed before application of changes, hence this mode copes better with external processes updating the objects.

The (version of) data is checked when data is flushed (typically at commit) to ensure the integrity in this respect. The most convenient way of checking data for updates is to maintain a column on each table that handles optimistic locking data to store a version.

Rather than placing version columns on all user datastore tables, Jakarta allows the user to notate particular classes as requiring *optimistic* treatment. This is performed by specifying in MetaData (XML or annotations) the details of the field/column to use for storing the version - see versioning. With strict Jakarta Persistence you must have a field in your class ready to store the version. With DataNucleus we also allow a version to be stored in a surrogate column hence not requiring a field in the actual class.

In terms of the process of optimistic locking, we demonstrate this below.

Operation	DataNucleus process	Datastore process
Start transaction		
Persist object	Prepare object (1) for persistence	
Update object	Prepare object (2) for update	
Persist object	Prepare object (3) for persistence	
Update object	Prepare object (4) for update	

Operation	DataNucleus process	Datastore process
Flush	Flush all outstanding changes to the datastore	• **Open connection** • Version check of object (1) • Insert the object (1) in the datastore. • Version check of object (2) • Update the object (2) in the datastore. • Version check of object (3) • Insert the object (3) in the datastore. • Version check of object (4) • Update the object (4) in the datastore.
Perform query	Generate query in datastore language	Query the datastore and return selected objects
Persist object	Prepare object (5) for persistence	
Update object	Prepare object (6) for update	
Commit transaction	Flush all outstanding changes to the datastore	• Version check of object (5) • Insert the object (5) in the datastore • Version check of object (6) • Update the object (6) in the datastore. • **Commit connection**

We have our flush mode set to not commit until flush/commit is called (FlushModeType.AUTO). When flush is performed (either manually, via commit, or via a query requiring it) the version check(s) are performed for any modified objects to be flushed, as long as they have a version defined. Please note that for some datastores (e.g RDBMS) the version check followed by update/delete is performed in a single statement.

See also :-

- Jakarta MetaData reference for <version> element
- Jakarta Annotations reference for @Version

Pessimistic (Datastore) Locking

Pessimistic locking isn't the default behaviour with Jakarta but can be configured. It is suitable for short lived operations where no user interaction is taking place and so it is possible to block access to datastore entities for the duration of the transaction. Such locking is best employed on specific objects, rather as a global process applying to all retrieved objects.

To disable optimistic locking (or version checking) globally you would add the persistence property **datanucleus.Optimistic** as *false*. Additionally, for RDBMS, to pessimistically lock ALL retrieved objects you would also set the persistence property **datanucleus.rdbms.useUpdateLock** to *true*.

Any object that has a pessimistic lock will result in (for RDBMS) all "SELECT ... FROM ..." retrieval statements being changed to be "SELECT ... FROM ... FOR UPDATE"; this will be applied only where the underlying RDBMS supports the "FOR UPDATE" syntax.

With pessimistic locking DataNucleus will grab a datastore connection at the first operation, and maintain it for the duration of the transaction. A single connection is used for the transaction (with the exception of any Value Generation operations which need datastore access, so these can use their own connection).

The Jakarta EntityManager allows control over locking on an object-by-object basis with several methods. For example

```
Person person = em.find(Person.class, 1, LockModeType.PESSIMISTIC_READ);
```

will retrieve the *Person* object with identity 1, and will lock it until the end of the transaction.

You can additionally perform an explicit lock on a specific object like this

```
em.lock(person, LockModeType.PESSIMISTIC_READ);
```

which will lock the object from that point in the transaction.

If you wanted to lock all objects affected by a query, you can set the lock mode of the query, like this

```
Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = 'Smith'");
q.setLockMode(LockModeType.PESSIMISTIC_READ);
List<Person> results = q.getResultList();
```

which will add a pessimistic lock on all *Person* objects with surname *Smith*.

In terms of the process of pessimistic (datastore) locking, we demonstrate this below. Here we have disabled the default "optimistic" check process (using **datanucleus.Optimistic** being set to *false*).

Operation	DataNucleus process	Datastore process
Start transaction		
Persist object	Prepare object (1) for persistence	**Open connection.** Insert the object (1) into the datastore
Update object	Prepare object (2) for update	Update the object (2) into the datastore
Persist object	Prepare object (3) for persistence	Insert the object (3) into the datastore

Operation	DataNucleus process	Datastore process
Update object	Prepare object (4) for update	Update the object (4) into the datastore
Flush	No outstanding changes so do nothing	
Perform query	Generate query in datastore language	Query the datastore and return selected objects
Persist object	Prepare object (5) for persistence	Insert the object (5) into the datastore
Update object	Prepare object (6) for update	Update the object (6) into the datastore
Commit transaction		**Commit connection**

So here (due to the flush mode chosen, and due to the default optimistic locking being disabled), whenever an operation is performed, DataNucleus pushes it straight to the datastore. Consequently any queries will always reflect the current state of all objects in use. This mode of operation has no version checking of objects and so, if they were updated by external processes in the meantime then, they will overwrite those changes. This is where the locking statements for particular objects is crucial, preventing them being updated externally.

One further thing to note is that you can have optimistic locking, whilst also having pessimistic locking of specific objects. You achieve this by following the optimistic locking process above, but using *find* and *createQuery* to lock specific objects using an appropriate pessimistic `LockModeType`.

It should be noted that DataNucleus provides two persistence properties that allow an amount of control over when flushing happens with pessimistic locking

- *datanucleus.flush.mode* when set to MANUAL will try to delay all datastore operations until commit/flush.
- *datanucleus.datastoreTransactionFlushLimit* represents the number of dirty objects before a flush is performed. This defaults to 1.

Datastore Connections

DataNucleus utilises datastore connections as follows

- EMF : single connection at any one time for datastore-based value generation. Obtained just for the operation, then released
- EMF : single connection at any one time for schema-generation. Obtained just for the operation, then released
- EM : single connection at any one time. When in a transaction the connection is held from the point of retrieval until the transaction commits or rolls back. The exact point at which the connection is obtained is defined more fully below. When used for non-transactional operations the connection is obtained just for the specific operation (unless configured to retain it).

If you are performing any schema generation at runtime then you must define a secondary connection factory (via use of *jakarta.persistence.jdbc.url*, or via *non-jta-datasource*).

If you have multiple threads using the same *EntityManager* then you can get "ConnectionInUse" problems where another operation on another thread comes in and tries to perform something while that first operation is still in use. This happens because the Jakarta Persistence spec requires an implementation to use a single datastore connection at any one time. When this situation crops up the user ought to use multiple *EntityManagers*.

Another important aspect is use of queries for Optimistic transactions, or for non-transactional contexts. In these situations it isn't possible to keep the datastore connection open indefinitely and so when the *Query* is executed the ResultSet is then read into memory making the queried objects available thereafter.

Transactional Context

For pessimistic/datastore transactions a connection will be obtained from the datastore when the first persistence operation is initiated. This datastore connection will be held **for the duration of the transaction** until such time as either *commit()* or *rollback()* are called.

For optimistic transactions the connection is only obtained when flush()/commit() is called. When flush() is called, or the transaction committed a datastore connection is finally obtained and it is held open until commit/rollback completes. When a datastore operation is required, the connection is typically released after performing that operation. So datastore connections, in general, are held for much smaller periods of time. This is complicated slightly by use of the persistence property **datanucleus.IgnoreCache**. When this is set to *false*, the connection, once obtained, is not released until the call to commit()/rollback().

For Neo4j/MongoDB a single connection is used for the duration of the EM for all transactional and nontransactional operations.

Nontransactional Context

When performing non-transactional operations, the default behaviour is to obtain a connection when needed, and release it after use. With RDBMS you have the option of retaining this connection ready for the next operation to save the time needed to obtain it; this is enabled by setting the persistence property **datanucleus.connection.nontx.releaseAfterUse** to *false*.

For Neo4j/MongoDB a single connection is used for the duration of the EM for all transactional and nontransactional operations.

Single Connection Mode

By default the connection used for transactional and non-transactional operations will be different, potentially from a different connection factory. If you set persistence property **datanucleus.connection.singleConnectionPerExecutionContext** to *true* then the connection for both transactional and non-transactional will come from the primary factory only. In addition, any connection from a transaction will not be released after commit of the transaction, and will be used thereafter for any non-transactional operations, as well as further transactions within the same EM context.

User Connection

DataNucleus provides a mechanism for users to access the native connection to the datastore, so that they can perform other operations as necessary. You obtain a connection as follows

```
// Obtain the connection from the Jakarta implementation
NucleusConnection ec = em.unwrap(NucleusConnection.class);
try
{
    Object native = conn.getNativeConnection();
    // Cast "native" to the required type for the datastore, see below

    ... use the connection to perform some operations.
}
finally
{
    // Hand the connection back to Jakarta
    conn.close();
}
```

For the datastores supported by DataNucleus, the "native" object is of the following types

- RDBMS : `java.sql.Connection`
- Excel : `org.apache.poi.hssf.usermodel.HSSFWorkbook`
- OOXML : `org.apache.poi.hssf.usermodel.XSSFWorkbook`
- ODF : `org.odftoolkit.odfdom.doc.OdfDocument`

- LDAP : `javax.naming.ldap.LdapContext`
- MongoDB : `com.mongodb.DB`
- XML : `org.w3c.dom.Document`
- Neo4j : `org.neo4j.graphdb.GraphDatabaseService`
- Cassandra : `com.datastax.driver.core.Session`
- HBase : NOT SUPPORTED
- JSON : NOT SUPPORTED
- *NeoDatis : org.neodatis.odb.ODB*
- *GAE Datastore : com.google.appengine.api.datastore.DatastoreService*

You **must** return the connection back to the *EntityManager* before performing any EntityManager operation. You do this by calling *conn.close()*. If you don't return the connection and try to perform an *EntityManager* operation which requires the connection then an exception is thrown.

Connection Pooling

When you create an *EntityManagerFactory* using the connection URL, driver name and the username/password to use, this doesn't necessarily pool the connections (so they would be efficiently opened/closed when needed to utilise datastore resources in an optimum way). For some of the supported datastores DataNucleus allows you to utilise a connection pool to efficiently manage the connections to the datastore. We currently provide support for the following

- RDBMS : Apache DBCP v2, we allow use of externally-defined DBCP2, but also provide a builtin DBCP v2.x
- RDBMS : C3P0
- RDBMS : BoneCP
- RDBMS : HikariCP
- RDBMS : Tomcat
- RDBMS : Manually creating a DataSource for a 3rd party software package
- RDBMS : Custom Connection Pooling Plugins for RDBMS using the DataNucleus ConnectionPoolFactory interface
- RDBMS : Using JNDI, and lookup a connection DataSource.
- LDAP : Using JNDI

You need to specify the persistence property **datanucleus.connectionPoolingType** to be whichever of the external pooling libraries you wish to use (or "None" if you explicitly want no pooling). DataNucleus provides two sets of connections to the datastore - one for transactional usage, and one for non-transactional usage. If you want to define a different pooling for nontransactional usage then you can also specify the persistence property **datanucleus.connectionPoolingType.nontx** to whichever is required.

RDBMS : JDBC driver properties with connection pool

If using RDBMS and you have a JDBC driver that supports custom properties, you can still use DataNucleus connection pooling and you need to s pecify the properties in with your normal persistence properties, but add the prefix **datanucleus.connectionPool.driver.** to the property name that the driver requires. For example, if an Oracle JDBC driver accepts *defaultRowPrefetch*, then you would specify something like

```
datanucleus.connectionPool.driver.defaultRowPrefetch=50
```

and it will pass in *defaultRowPrefetch* as "50" into the driver used by the connection pool.

RDBMS : Apache DBCP v2+

DataNucleus provides a builtin version of DBCP2 to provide pooling. This is automatically selected if using RDBMS, unless you specify otherwise. An alternative is to use an external DBCP2. This is accessed by specifying the persistence property **datanucleus.connectionPoolingType** to *DBCP2* in your `persistence.xml`.

So the *EMF* will use connection pooling using DBCP version 2. To do this you will need `commons-dbcp2`, `commons-pool2` JARs to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for DBCP2 are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxIdle=10
datanucleus.connectionPool.minIdle=3
datanucleus.connectionPool.maxActive=5
datanucleus.connectionPool.maxWait=60

datanucleus.connectionPool.testSQL=SELECT 1

datanucleus.connectionPool.timeBetweenEvictionRunsMillis=2400000
```

RDBMS : C3P0

DataNucleus allows you to utilise a connection pool using C3P0 to efficiently manage the connections to the datastore. C3P0 is a third-party library providing connection pooling. This is accessed by specifying the persistence property **datanucleus.connectionPoolingType** to *C3P0* in your `persistence.xml`.

So the *EMF* will use connection pooling using C3P0. To do this you will need the `c3p0` JAR to be in the CLASSPATH.

If you want to configure C3P0 further you can include a `c3p0.properties` in your CLASSPATH - see the C3P0 documentation for details. You can also specify persistence properties to control the actual pooling. The currently supported properties for C3P0 are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxPoolSize=5
datanucleus.connectionPool.minPoolSize=3
datanucleus.connectionPool.initialPoolSize=3

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=20
```

RDBMS : BoneCP

DataNucleus allows you to utilise a connection pool using BoneCP to efficiently manage the connections to the datastore. BoneCP is a third-party library providing connection pooling. This is accessed by specifying the persistence property **datanucleus.connectionPoolingType** to *BoneCP* in your `persistence.xml`.

So the *EMF* will use connection pooling using BoneCP. To do this you will need the `bonecp` JAR (and `slf4j`, `google-collections`) to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for BoneCP are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxPoolSize=5
datanucleus.connectionPool.minPoolSize=3

# Pooling of PreparedStatements
datanucleus.connectionPool.maxStatements=20
```

RDBMS : HikariCP

DataNucleus allows you to utilise a connection pool using HikariCP to efficiently manage the connections to the datastore. HikariCP is a third-party library providing connection pooling. This is accessed by specifying the persistence property **datanucleus.connectionPoolingType** to *HikariCP* in your `persistence.xml`.

So the *EMF* will use connection pooling using HikariCP. To do this you will need the `hikaricp` JAR (and `slf4j`, `javassist` as required) to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling. The currently supported properties for HikariCP are shown below

```
# Pooling of Connections
datanucleus.connectionPool.maxPoolSize=5
datanucleus.connectionPool.idleTimeout=200
datanucleus.connectionPool.leakThreshold=1
datanucleus.connectionPool.maxLifetime=240
```

RDBMS : Tomcat

DataNucleus allows you to utilise a connection pool using Tomcat JDBC Pool to efficiently manage the connections to the datastore. This is accessed by specifying the persistence property **datanucleus.connectionPoolingType** to *tomcat* in your `persistence.xml`.

So the *EMF* will use a DataSource with connection pooling using Tomcat. To do this you will need the `tomcat-jdbc` JAR to be in the CLASSPATH.

You can also specify persistence properties to control the actual pooling, like with the other pools.

RDBMS : Manually create a DataSource ConnectionFactory

We could have used the built-in DBCP2 support which internally creates a DataSource ConnectionFactory, alternatively the support for external DBCP, C3P0, HikariCP, BoneCP etc, however we can also do this manually if we so wish. Let's demonstrate how to do this with one of the most used pools Apache Commons DBCP

With DBCP you need to generate a **javax.sql.DataSource**, which you will then pass to DataNucleus. You do this as follows

```
// Load the JDBC driver
Class.forName(dbDriver);

// Create the actual pool of connections
ObjectPool connectionPool = new GenericObjectPool(null);

// Create the factory to be used by the pool to create the connections
ConnectionFactory connectionFactory = new DriverManagerConnectionFactory(dbURL,
dbUser, dbPassword);

// Create a factory for caching the PreparedStatements
KeyedObjectPoolFactory kpf = new StackKeyedObjectPoolFactory(null, 20);

// Wrap the connections with pooled variants
PoolableConnectionFactory pcf =
    new PoolableConnectionFactory(connectionFactory, connectionPool, kpf, null, false,
true);

// Create the datasource
DataSource ds = new PoolingDataSource(connectionPool);

// Create our EMF
Map properties = new HashMap();
properties.put("datanucleus.ConnectionFactory", ds);
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit",
properties);
```

Note that we haven't passed the *dbUser* and *dbPassword* to the EMF since we no longer need to

specify them - they are defined for the pool so we let it do the work. As you also see, we set the data source for the EMF. Thereafter we can sit back and enjoy the performance benefits. Please refer to the documentation for DBCP for details of its configurability (you will need `commons-dbcp`, `commons-pool`, and `commons-collections` in your CLASSPATH to use this above example).

RDBMS : Lookup a DataSource using JNDI

DataNucleus allows you to use connection pools (java.sql.DataSource) bound to a **javax.naming.InitialContext** with a JNDI name. You first need to create the DataSource in the container (application server/web server), and secondly you specify the *jta-data-source* in the persistence-unit with the DataSource JNDI name. Please read more about this in RDBMS DataSources.

LDAP : JNDI

If using an LDAP datastore you can use the following persistence properties to enable connection pooling

```
datanucleus.connectionPoolingType=JNDI
```

Once you have turned connection pooling on if you want more control over the pooling you can also set the following persistence properties

- **datanucleus.connectionPool.maxPoolSize** : max size of pool
- **datanucleus.connectionPool.initialPoolSize** : initial size of pool

Data Sources

Applicable to RDBMS

DataNucleus allows use of a *data source* that represents the datastore in use. With Jakarta Persistence you specify this typically as the JNDI name of the datasource location. This is often just a URL defining the location of the datastore, but there are in fact several ways of specifying this *data source* depending on the environment in which you are running.

- Nonmanaged Context - Java Client
- Managed Context - Servlet
- Managed Context - JavaEE

Java Client Environment : Non-managed Context

DataNucleus permits you to take advantage of using database connection pooling that is available on an application server. The application server could be a full JEE server (e.g WebLogic) or could equally be a servlet engine (e.g Tomcat, Jetty). Here we are in a non-managed context, and we use the following properties when creating our EntityManagerFactory, and refer to the JNDI data source of the server.

If the data source is available in WebLogic, the simplest way of using a data source outside the application server is as follows.

```
Map ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL,"t3://localhost:7001");
Context ctx = new InitialContext(ht);
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("datanucleus.ConnectionFactory",ds);
EntityManagerFactory emf = ...
```

If the data source is available in Websphere, the simplest way of using a data source outside the application server is as follows.

```
Map ht = new Hashtable();
ht.put(Context.INITIAL_CONTEXT_FACTORY,"com.ibm.websphere.naming.WsnInitialContextFact
ory");
ht.put(Context.PROVIDER_URL,"iiop://server:orb port");

Context ctx = new InitialContext(ht);
DataSource ds = (DataSource) ctx.lookup("jdbc/datanucleus");

Map properties = new HashMap();
properties.setProperty("datanucleus.ConnectionFactory",ds);
EntityManagerFactory emf = ...
```

Servlet Environment : Managed Context

As an example of setting up such a JNDI data source for Tomcat 5.0, here we would add the following file to *$TOMCAT/conf/Catalina/localhost/* as `datanucleus.xml`

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="/home/datanucleus/" path="/datanucleus">
    <Resource name="jdbc/datanucleus" type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/datanucleus">
        <parameter>
            <name>maxWait</name>
            <value>5000</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>20</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>2</value>
        </parameter>

        <parameter>
            <name>url</name>
            <value>jdbc:mysql://127.0.0.1:3306/datanucleus?autoReconnect=true</value>
        </parameter>
        <parameter>
            <name>driverClassName</name>
            <value>com.mysql.jdbc.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>mysql</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value></value>
        </parameter>
    </ResourceParams>
</Context>
```

With this Tomcat JNDI data source we would then specify the data source (name) as *java:comp/env/jdbc/datanucleus*.

```
Properties properties = new Properties();
properties.setProperty("jakarta.persistence.jtaDataSource","java:comp/env/jdbc/datanuc
leus");
EntityManagerFactory emf = ...
```

JavaEE : Managed Context

As in the above example, we can also run in a managed context, in a JavaEE/Servlet environment, and here we would make a minor change to the specification of the JNDI data source depending on

the application server or the scope of the *jndi:* global or component.

Using JNDI deployed in global environment:

```
Properties properties = new Properties();
properties.setProperty("jakarta.persistence.jtaDataSource","jdbc/datanucleus");
EntityManagerFactory emf = ...
```

Using JNDI deployed in component environment:

```
Properties properties = new Properties();
properties.setProperty("jakarta.persistence.jtaDataSource","java:comp/env/jdbc/datanuc
leus");
EntityManagerFactory emf = ...
```

Multitenancy

On occasion you need to share a data model with other user-groups or other applications and where the model is persisted to the same structure of datastore. There are three ways of handling this with DataNucleus.

- **Separate Database per Tenant** - have a different database per user-group/application. In this case you will have a separate EMF for each database, and manage use of the appropriate EMF yourself.
- **Separate Schema per Tenant** - as the first option, except use different schemas. In this case you will have a separate EMF for each database schema, and manage use of the appropriate EMF yourself.
- **Same Database/Schema but with a Discriminator in affected Table(s)** - this is described below. In this case you will have a single EMF, and DataNucleus will manage selecting appropriate data for the tenant in question. This is described below.

Multitenancy via Discriminator in Table

Applicable to RDBMS, HBase, MongoDB, Neo4j, Cassandra

If you specify the persistence property **datanucleus.tenantId** as an identifier for your user-group/application then DataNucleus will know that it needs to provide a tenancy discriminator to all primary tables of persisted classes. This discriminator is then used to separate the data of the different user-groups.

The **Tenant ID** can be set in one of three ways.

- Per EntityManagerFactory : just set the persistence property **datanucleus.tenantId** when you start up the EMF, and all access for this EMF will use this Tenant ID
- Per EntityManager : set the persistence property **datanucleus.tenantId** when you start up the EMF as the default Tenant ID, and set a property on any EntityManager that you want a different Tenant ID specifying for. Like this

```
EntityManager em = emf.createEntityManager();
... // All operations will apply to default tenant specified in persistence property
for EMF
em.close();

EntityManager em1 = emf.createEntityManager();
em1.setProperty("datanucleus.tenantId", "John");
... // All operations will apply to tenant "John"
em1.close();

EntityManager em2 = emf.createEntityManager();
em2.setProperty("datanucleus.tenantId", "Chris");
... // All operations will apply to tenant "Chris"
em2.close();
```

- Per datastore access : When creating the EMF set the persistence property **datanucleus.tenantProvider** and set it to an instance of *org.datanucleus.store.schema.MultiTenancyProvider* Javadoc

```
public interface MultiTenancyProvider
{
    String getTenantId(ExecutionContext ec);
}
```

Now the programmer can set a different Tenant ID for each datastore access, maybe based on some session variable for example?.

Read access to data from multiple tenants

Applicable to RDBMS

An additional flexibility for reading data from the datastore, you may want read access to the data of particular tenants. To allow this you can set the persistence property **datanucleus.TenantReadIds** to a comma separated list of the tenant ids to read from. This can only be set on the EMF. Any data written with this EMF will still use the *tenantId* defined earlier.

Bean Validation

Support for BeanValidation includes all versions of that API (1.0, 1.1, 2.0).

The Bean Validation API (JSR0303/JSR0349/JSR0380) can be hooked up with Jakarta Persistence so that you have validation of an objects values prior to persistence, update and deletion. To do this

- Put the **javax.validation** `validation-api` jar in your CLASSPATH, along with the Bean Validation implementation jar of your choice (e.g Apache BVal)
- Set the persistence property **jakarta.persistence.validation.mode** to one of *auto* (default), *none*, or *callback*
- Optionally set the persistence property(s) **jakarta.persistence.validation.group.pre-persist**, **jakarta.persistence.validation.group.pre-update**, **jakarta.persistence.validation.group.pre-remove** to fine tune the behaviour (the default is to run validation on pre-persist and pre-update if you don't specify these).
- Use Jakarta Persistence as you normally would for persisting objects

To give a simple example of what you can do with the Bean Validation API

```
@Entity
public class Person
{
    @Id
    @NotNull
    private Long id;

    @NotNull
    @Size(min = 3, max = 80)
    private String name;

    ...
}
```

So we are validating that instances of the *Person* class will have an "id" that is not null and that the "name" field is not null and between 3 and 80 characters. If it doesn't validate then at persist/update an exception will be thrown. You can add bean validation annotations to classes marked as `@Entity`, `@MappedSuperclass` or `@Embeddable`.

A further use of the Bean Validation annotations `@Size(max=…)` and `@NotNull` is that if you specify these then you have no need to specify the equivalent Jakarta attributes since they equate to the same thing. This is enabled via the persistence property **datanucleus.metadata.javaxValidationShortcuts**.

Entity Graphs

When an object is retrieved from the datastore by Jakarta Persistence typically not all fields are retrieved immediately. This is because for efficiency purposes only particular field types are retrieved in the initial access of the object, and then any other objects are retrieved when accessed (lazy loading). The group of fields that are loaded is called an **entity graph**. There are 3 types of "entity graphs" to consider

- Default Entity Graph : implicitly defined in all Jakarta Persistence specs, specifying the *fetch* setting for each field/property (LAZY/EAGER).
- Named Entity Graphs : allows the user to define *Named Entity Graphs* in metadata, via annotations or XML
- Unnamed Entity Graphs : allows the user to define Entity Graphs via the Jakarta API at runtime

Default Entity Graph

Jakarta Persistence provides an initial entity graph, comprising the fields that will be retrieved when an object is retrieved if the user does nothing to define the required behaviour. You define this "default" by setting the *fetch* attribute in metadata for each field/property.

Named Entity Graphs

You can predefine **Named Entity Graphs** in metadata which can then be used at runtime when retrieving objects from the datastore (via find/query). For example, if we have the following class

```
class MyClass
{
    String name;
    Set coll;
    MyOtherClass other;
}
```

and we want to have the option of the *other* field loaded whenever we load objects of this class, we define our annotations as

```
@Entity
@NamedEntityGraph(name="includeOther", attributeNodes={@NamedAttributeNode("other")})
public class MyClass
{
    ...
}
```

So we have defined an EntityGraph called "includeOther" that just includes the field with name *other*. We can retrieve this and then use it in our persistence code, as follows

```
EntityGraph includeOtherGraph = em.getEntityGraph("includeOther");

Properties props = new Properties();
props.put("jakarta.persistence.loadgraph", includeOtherGraph);
MyClass myObj = em.find(MyClass.class, id, props);
```

Here we have made use of the *EntityManager.find* method and provided the property **jakarta.persistence.loadgraph** to be our EntityGraph. This means that it will fetch all fields in the *default* EntityGraph, **plus** all fields in the *includeOther* EntityGraph. If we had provided the property **jakarta.persistence.fetchgraph** set to our EntityGraph it would have fetched just the fields defined in that EntityGraph.

Note that you can also make use of EntityGraphs when using the Jakarta Query API, specifying the same properties above but as query *hints*.

Unnamed Entity Graphs

You can define **Entity Graphs** at runtime, programmatically. For example, if we have the following class

```
class MyClass
{
    String name;
    HashSet coll;
    MyOtherClass other;
}
```

and we want to have the option of the *other* field loaded whenever we load objects of this class, we do the following

```
EntityGraph includeOtherGraph = em.createEntityGraph(MyClass.class);
includeOtherGraph.addAttributeNodes("other");
```

So we have defined an EntityGraph that just includes the field with name *other*. We can then use this at runtime in our persistence code, as follows

```
Properties props = new Properties();
props.put("jakarta.persistence.loadgraph", includeOtherGraph);
MyClass myObj = em.find(MyClass.class, id, props);
```

Here we have made use of the *EntityManager.find* method and provided the property **jakarta.persistence.loadgraph** to be our EntityGraph. This means that it will fetch all fields in the *default* EntityGraph, **plus** all fields in this EntityGraph. If we had provided the property **jakarta.persistence.fetchgraph** set to our EntityGraph it would have fetched just the fields defined in that EntityGraph.

Note that you can also make use of EntityGraphs when using the Jakarta Query API, specifying the same properties above but as query *hints*, like this

```
EntityGraph<MyClass> eg = em.createEntityGraph(MyClass.class);
eg.addAttributeNodes("id");
eg.addAttributeNodes("name");
eg.addAttributeNodes("other");
Subgraph<MyOtherClass> myOtherClassGraph = eg.addSubgraph("other", MyOtherClass.
class);
myOtherClass.addAttributeNodes("name");

Query q = em.createQuery("SELECT m FROM MyClass m");
q.setHint("jakarta.persistence.fetchgraph", eg);
List<MyClass> results = q.getResultList();
```

Lifecycle Callbacks

Jakarta Persistence defines a mechanism whereby an Entity can be marked as a listener for lifecycle events. Alternatively a separate entity listener class can be defined to receive these events. Thereafter when entities of the particular class go through lifecycle changes events are passed to the provided methods. Let's look at the two different mechanisms

Entity Callbacks

An Entity itself can have several methods defined to receive events when any instances of that class pass through lifecycles changes. Let's take an example

```
@Entity
public class Account
{
    @Id
    Long accountId;

    Integer balance;
    boolean preferred;

    public Integer getBalance() { ... }

    @PrePersist
    protected void validateCreate()
    {
        if (getBalance() < MIN_REQUIRED_BALANCE)
        {
            throw new AccountException("Insufficient balance to open an account");
        }
    }

    @PostLoad
    protected void adjustPreferredStatus()
    {
        preferred = (getBalance() >= AccountManager.getPreferredStatusLevel());
    }
}
```

So in this example just before any "Account" object is persisted the *validateCreate* method will be called. In the same way, just after the fields of any "Account" object are loaded the *adjustPreferredStatus* method is called. Very simple.

You can register callbacks for the following lifecycle events

- PrePersist
- PostPersist

- PreRemove
- PostRemove
- PreUpdate
- PostUpdate
- PostLoad

The only other rule is that any method marked to be a callback method has to take no arguments as input, and have void return.

Entity Listener

As an alternative to having the actual callback methods in the Entity class itself you can define a separate class as an *EntityListener*. So lets take the example shown before and do it for an EntityListener.

```
@Entity
@EntityListeners(mydomain.MyEntityListener.class)
public class Account
{
    @Id
    Long accountId;

    Integer balance;
    boolean preferred;

    public Integer getBalance() { ... }
}
```

```
package mydomain;

public class MyEntityListener
{
    @PostPersist
    public void newAccountAlert(Account acct)
    {
         ... do something when we get a new Account
    }
}
```

So we define our "Account" entity as normal but mark it with an *EntityListener*, and then in the *EntityListener* we define the callbacks we require. As before we can define any of the 7 callbacks as we require. The only difference is that the callback method has to take an argument of type "Object" that it will be called for, and have void return.

The Entity Listeners objects shown here are **stateless**.

DataNucleus allows for stateful event listener objects, with the state being CDI injectable, but you must be in a CDI environment for this to work. To provide CDI support for Jakarta, you should specify the persistence property **jakarta.persistence.bean.manager** to be a CDI `BeanManager` object.

JavaEE Environments

Jakarta Persistence is designed to allow easy deployment into a JavaEE container. The JavaEE container takes care of integration of the Jakarta implementation (DataNucleus), so there is no JCA connector required.

Key points to remember when deploying your Jakarta application to use DataNucleus under JavaEE

- Define a JTA datasource for your persistence operations
- Define a non-JTA datasource for your schema and sequence operations. These are cross-EntityManager and so need their own datasource that is not affected by transactions.

Individual guides for specific JavaEE servers are listed below. If you have a guide for some other server, please notify us and it will be added to this list.

JBoss AS7

This guide was provided by Nicolas Seyvet. It is linked to from the JBoss docs.

JBoss AS7 is a recent JavaEE server from JBoss. Despite searching in multiple locations, I could not find a comprehensive guide on how to switch from the default JBoss Hibernate JPA provider to Datanucleus. If you try this guide, please PM the author (or add a comment) and let me know how it worked out. Your feedback will be used to improve this guide. This guide is cross-referenced as part of the JBoss JPA Reference Guide.

JBossAS7 : Download JBoss AS7 and DataNucleus

JBoss : At the time I am writing this "How To", the latest JBoss AS available from the main JBoss community site is 7.1.1.Final aka Brontes. In this guide, the latest 7.x SNAPSHOT was used but the steps will work with any JBoss 7.x version.

DataNucleus : Version 5.0 was used, from SourceForge but should work with later versions.

JBossAS7 : Install JBoss AS 7

Install JBoss AS 7 by unzipping the downloaded JBoss zip file in the wanted folder to be used as the JBoss home root folder (example: /local/jboss). From this point, the path where JBoss is unzipped will be referred to as **$JBOSS_HOME**.

Note: JBoss AS 7 configuration is controlled by either `standalone.xml` (*$JBOSS_HOME/standalone/configuration*) or `domain.xml` (*$JBOSS_HOME/domain/configuration*) depending on the operation mode (standalone or domain) of the application server. The domain mode is typically used for cases where the AS is deployed in a cluster environment. In this tutorial, a single AS instance is used, as such, the standalone mode is selected and all configuration changes will be applied to the `standalone.xml` file.

JBossAS7 : Start JBoss

To start the server, use:

```
On Linux:

$ cd $JBOSS_HOME/bin/
$ ./standalone.sh

On Windows:

$ cd $JBOSS_HOME/bin/
$ standalone.bat
```

After a few seconds, a message should indicate the server is started.

```
17:23:00,251 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874: JBoss AS
7.2.0.Alpha1-SNAPSHOT "Steropes" started
 in 3717ms - Started 198 of 257 services (56 services are passive or on-demand)
```

To verify, access the administration GUI located at http://localhost:9990/, and expect to see a "Welcome to AS 7" banner. On the first start up, a console will show that an admin user must first be created in order to be able to access the management UI. Follow the steps and create a user.

```
On Linux:

$JBOSS_HOME/bin$ add-user.sh


On Windows:

$JBOSS_HOME/bin$ add-user.bat
```

JBossAS7 : Add a JDBC DataSource (Optional)

This step is only necessary if an RDBMS solution is used as a data store, or if external drivers are required. This tutorial will use MySQL as the RDBMS storage, and the required drivers and data source will be added. For more information, about data sources under JBoss AS 7, refer to the JBoss docs

JBossAS7 : Add MySQL drivers

For MySQL, it is recommended to use Connector/J, which can be found here. This tutorial uses version 5.1.20.

JBoss uses OSGI to define a set of modules, further info about class loading in JBoss. In short, the configuration files binds the services and the modules, defining what is available in the class loader for a specific service or application.

While dropping the drivers in the *$JBOSS_HOME/standalone/deployments* directory works, this approach is not recommended. The proper approach is to add the drivers by defining a new module containing the required libraries. The full instructions are available under here.

Short walk through for MySQL:

- Get the drivers
- create a "mysql" directory under *$JBOSS_HOME/modules/com/*
- create a "main" directory under *$JBOSS_HOME/modules/com/mysql*
- Copy the "mysql-connector-java-5.1.20-bin.jar" drivers under *$JBOSS_HOME/modules/com/mysql/main*
- Add a `module.xml` file under *$JBOSS_HOME/modules/com/mysql/main*

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
    <resources>
        <resource-root path="mysql-connector-java-5.1.20-bin.jar"/>
    </resources>
    <dependencies>
        <module name="javax.api"/>
    </dependencies>
</module>
```

The **name** is important as it defines the module name and is used in the `standalone.xml` configuration file. Now, let's say the URL to the MySQL database to be used is "jdbc:mysql://localhost:3306/simple", there are three ways to add that to the server, either through the management console at localhost or, by modifying the `standalone.xml` configuration file, or by using the Command Line Interface (CLI).

Let's modify the `standalone.xml` file. Verify the AS is stopped. Open `standalone.xml` for editing. Search for "subsystem xmlns="urn:jboss:domain:datasources:1.1", the section defines data sources and driver references. Let's add our data source and drivers. Add the following in the **datasources** section:

```
<datasource jndi-name="java:/jdbc/simple" pool-name="MySQL-DS" enabled="true">
    <connection-url>jdbc:mysql://localhost:3306/simple</connection-url>
    <driver>com.mysql</driver>
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-isolation>
    <pool>
        <min-pool-size>10</min-pool-size>
        <max-pool-size>100</max-pool-size>
        <prefill>true</prefill>
    </pool>
    <security>
        <user-name>[A valid DB user name]</user-name>
        <password>[A valid DB password]</password>
    </security>
    <statement>
        <prepared-statement-cache-size>32</prepared-statement-cache-size>
        <share-prepared-statements>true</share-prepared-statements>
    </statement>
</datasource>
<datasource jta="false" jndi-name="java:/jdbc/simple-nonjta" pool-name="MySQL-DS-
NonJTA" enabled="true">
    <connection-url>jdbc:mysql://localhost:3306/simple</connection-url>
    <driver>com.mysql</driver>
    <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-isolation>
    <security>
        <user-name>[A valid DB user name]</user-name>
        <password>[A valid DB password]</password>
    </security>
    <statement>
        <share-prepared-statements>false</share-prepared-statements>
    </statement>
</datasource>
```

The above defines two data sources (MySQL-DS and MySQL-DS-NonJTA) referring to the same database. The difference between the two is that MySQL-DS has JTA enabled while MySQL-DS-NonJTA does not. This is useful to separate operations during the database automated schema generation phase. Any change to a schema should be made outside the scope of JTA. Many JDBC drivers (for example) will fall apart (assorted type of SQLException) if you try to commit a connection with DDL and SQL mixed, or SQL first then DDL after. Consequently it is recommended to have a separate data source for such operations, hence using the non-jta-data-source.

In the **drivers** section, add:

```
<driver name="com.mysql" module="com.mysql">
    <xa-datasource-class>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</xa-
datasource-class>
</driver>
```

The above defines which drivers to use for the data sources MySQL-DS and MySQL-DS-NonJTA.

More info is available as part of the JBoss documentation, refer to the section describing how to setup a new data source.

JBossAS7 : Add DataNucleus to JBoss

This step adds the DataNucleus libraries as a JBoss module.

- Create a directory to store the DataNucleus libraries, as **$JBOSS_HOME/modules/org/datanucleus/main**
- Add the following jars from the lib directory of the *datanucleus-accessplatform-full-deps* ZIP file *lib* directory : `datanucleus-api-jakarta-XXX.jar`, `datanucleus-core-XXX.jar`, `datanucleus-rdbms-XXX.jar`, `datanucleus-jakarta-query-XXX.jar`
- Add a `module.xml` file in the $JBOSS_HOME/modules/org/datanucleus/main directory like this

```
<module xmlns="urn:jboss:module:1.1" name="org.datanucleus">
    <dependencies>
        <module name="javax.api"/>
        <module name="jakarta.persistence.api"/>
        <module name="javax.transaction.api"/>
        <module name="javax.validation.api"/>
    </dependencies>
    <resources>
        <resource-root path="datanucleus-api-jakarta-6.0.0-m1.release.jar"/>
        <resource-root path="datanucleus-core-6.0.0-m1.release.jar"/>
        <resource-root path="datanucleus-rdbms-6.0.0-m1.release.jar"/>
        <resource-root path="datanucleus-jakarta-query-6.0.0-m1.release.jar"/>
    </resources>
</module>
```

At this point, all the Jakarta dependencies are resolved.

JBossAS7 : A simple example

Now you simply need to define `persistence.xml` and use Jakarta as you normally would. In order to use DataNucleus as a persistence provider, the `persistence.xml` file must contain the "jboss.as.jpa.providerModule" property. Using the datasources defined above, an example of a `persistence.xml` file could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="[Persistence Unit Name]" transaction-type="JTA">
        <provider>org.datanucleus.api.jakarta.PersistenceProviderImpl</provider>
        <!-- MySQL DS -->
        <jta-data-source>java:/jdbc/simple</jta-data-source>
        <non-jta-data-source>java:/jdbc/simple-nonjta</non-jta-data-source>

        <class>[Entities must be listed here]</class>

        <properties>
            <!-- Magic JBoss property for specifying the persistence provider -->
            <property name="jboss.as.jpa.providerModule" value="org.datanucleus"/>

            <!-- following is probably not useful... but it ensures we bind to the JTA
transaction manager...-->
            <property name="datanucleus.transaction.jta.transactionManagerLocator"
value="custom_jndi"/>
            <property name="datanucleus.transaction.jta.transactionManagerJNDI"
value="java:/TransactionManager"/>

            <property name="datanucleus.metadata.validate" value="false"/>
            <property name="datanucleus.schema.autoCreateAll" value="true"/>
            <property name="datanucleus.schema.validateTables" value="false"/>
            <property name="datanucleus.schema.validateConstraints" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

TomEE

Apache TomEE ships with OpenJPA/EclipseLink as the default JPA provider (depending on which version of TomEE), however any valid Jakarta provider can be used.

The basic steps are:

- Add the DataNucleus jars to *<tomee-home>/lib/*
- Configure the web-app or the server to use DataNucleus.

TomEE : Webapp Configuration

Any web-app can specify the Jakarta provider it would like to use via the `persistence.xml` file, which can be at any of the following locations in a web-app

- `WEB-INF/persistence.xml` of the .war file
- `META-INF/persistence.xml` in any jar located in WEB-INF/lib/

A single web-app may have many `persistence.xml` files and each may use whichever Jakarta

provider it needs. The following is an example of a fairly common `persistence.xml` for DataNucleus

```
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="movie-unit">
        <provider>org.datanucleus.api.jakarta.PersistenceProviderImpl</provider>
        <jta-data-source>movieDatabase</jta-data-source>
        <non-jta-data-source>movieDatabaseUnmanaged</non-jta-data-source>
        <properties>
            <property name="jakarta.persistence.schema-generation.database.action"
value="drop-and-create"/>
        </properties>
    </persistence-unit>
</persistence>
```

Note that you may have to set the persistence property **datanucleus.transaction.jta.transactionManagerLocator** and **datanucleus.transaction.jta.transactionManagerJNDI** to find your JTA "TransactionManager". See the persistence properties for details of those.

TomEE : Server Configuration

The default Jakarta provider can be changed at the server level to favour DataNucleus over OpenJPA/EclipseLink. Using the `<tomee-home>/conf/system.properties` file or any other valid means of setting java.lang.System.getProperties(), the following standard properties can set the default for any persistence.xml file.

```
jakarta.persistence.provider
jakarta.persistence.transactionType
jakarta.persistence.jtaDataSource
jakarta.persistence.nonJtaDataSource
```

So, for example, DataNucleus can become the default provider via setting

```
CATALINA_OPTS=-
Djakarta.persistence.provider=org.datanucleus.api.jakarta.PersistenceProviderImpl
```

You must of course add the DataNucleus libraries to *<tomee-home>/lib/* for this to work.

TomEE : DataNucleus libraries

Jars needed for DataNucleus 6.0:

Check for the latest release of each of these jars and update the versions accordingly.

```
# Add:
<tomee-home>/lib/datanucleus-core-6.0.0-m1.jar
<tomee-home>/lib/datanucleus-api-jakarta-6.0.0-m1.jar
<tomee-home>/lib/datanucleus-rdbms-6.0.0-m1.jar

# Remove (optional):
<tomee-home>/lib/asm-3.2.jar
<tomee-home>/lib/commons-lang-2.6.jar
<tomee-home>/lib/openjpa-2.2.0.jar (or EclipseLink)
<tomee-home>/lib/serp-1.13.1.jar
```

OSGi Environments

DataNucleus jars are OSGi bundles, and as such, can be deployed in an OSGi environment. Being an OSGi environment care must be taken with respect to class-loading. In particular the persistence property **datanucleus.primaryClassLoader** will need setting.

An important thing to note : any dependent jar that is required by DataNucleus needs to be OSGi enabled. By this we mean the jar needs to have the MANIFEST.MF file including *ExportPackage* for the packages required by DataNucleus. Failure to have this will result in *ClassNotFoundException* when trying to load its classes.

The `jakarta.persistence` jar that is included in the DataNucleus distribution is OSGi-enabled.

When using DataNucleus in an OSGi environment you can set the persistence property **datanucleus.plugin.pluginRegistryClassName** to *org.datanucleus.plugin.OSGiPluginRegistry*.

Jakarta and OSGi

In a non OSGi world the persitence provider implementation is loaded using the service provider pattern. The full qualified name of the implementation is stored in a file under *META-INF/services/jakarta.persistence.spi.PersistenceProvider* (inside the jar of the implementation) and each time the persistence provider is required it gets loaded with a *Class.forName* using the name of the implementing class found inside the *META-INF/services/jakarta.persistence.spi.PersistenceProvider*. In the OSGi world that doesn't work. The bundle that needs to load the persistence provider implementation cannot load `META-INF/services/jakarta.persistence.spi.PersistenceProvider`. A work around is to copy that file inside each bundle that requires access to the peristence provider. Another work around is to export the persistence provider as OSGi service. This is what the DataNucleus Jakarta API jar does.

Further reading available on this link

Sample using OSGi and Jakarta

Please make use of the OSGi sample. This provides a simple example that you can build and load into such as Apache Karaf to demonstrate JPA persistence. Here we attempt to highlight the key aspects specific to OSGi in this sample.

Model classes are written in the exact same way as you would for any application.

Creation of the EMF is specified in a persistence-unit as normal **except that** we need to provide two overriding properties

```
Map<Object, Object> overrideProps = new HashMap();
overrideProps.put("datanucleus.primaryClassLoader", this.getClass().getClassLoader());
overrideProps.put("datanucleus.plugin.pluginRegistryClassName",
"org.datanucleus.plugin.OSGiPluginRegistry");

EntityManagerFactory emf = Persistence.createEntityManagerFactory("PU",
overrideProps);
```

so we have provided a class loader for the OSGi context of the application, and also specified that we want to use the *OSGiPluginRegistry*.

All persistence and query operations using EntityManager etc thereafter are identical to what you would use in a normal JavaSE/JavaEE application.

The `pom.xml` also defines the imports/exports for our OSGi application bundle, so look at this if wanting guidance on what these could look like when using Maven and the "felix bundle" plugin.

If you read the file `README.txt` you can see basic instructions on how to deploy this application into a fresh download of Apache Karaf, and run it. It makes uses of Spring DM to start the JPA "application".

LocalContainerEntityManagerFactoryBean class for use in Virgo 3.0 OSGi environment

When using DataNucleus 3.x in a Virgo 3.0.x OSGi environment, which is essentially Eclipse Equinox + Spring dm Server with Spring 3.0.5.RELEASE included, the following class is working for me to use in your Spring configuration. You can use this class as a drop-in replacement for Spring's *org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean*. It was inspired by the code-ish sample at HOWTO Use Datanucleus with OSGi and Spring DM.

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

import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.PersistenceException;
import jakarta.persistence.spi.PersistenceUnitInfo;

import org.datanucleus.util.StringUtils;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.osgi.context.BundleContextAware;

public class DataNucleusOsgiLocalContainerEntityManagerFactoryBean extends
    LocalContainerEntityManagerFactoryBean implements BundleContextAware
{

    public static final String DEFAULT_JKA_API_BUNDLE_SYMBOLIC_NAME =
```

```
"org.datanucleus.api.jakarta";
    public static final String DEFAULT_PERSISTENCE_PROVIDER_CLASS_NAME =
"org.datanucleus.api.jakarta.PersistenceProviderImpl";

    public static final String DEFAULT_OSGI_PLUGIN_REGISTRAR_CLASS_NAME =
"org.datanucleus.plugin.OSGiPluginRegistry";
    public static final String DEFAULT_OSGI_PLUGIN_REGISTRAR_PROPERTY_NAME =
"datanucleus.plugin.pluginRegistryClassName";

    protected BundleContext bundleContext;
    protected ClassLoader classLoader;

    protected String jkaApiBundleSymbolicName = DEFAULT_JKA_API_BUNDLE_SYMBOLIC_NAME;
    protected String persistenceProviderClassName =
DEFAULT_PERSISTENCE_PROVIDER_CLASS_NAME;
    protected String osgiPluginRegistrarClassName =
DEFAULT_OSGI_PLUGIN_REGISTRAR_CLASS_NAME;
    protected String osgiPluginRegistrarPropertyName =
DEFAULT_OSGI_PLUGIN_REGISTRAR_PROPERTY_NAME;

    @Override
    public void setBundleContext(BundleContext bundleContext) {
    this.bundleContext = bundleContext;
    }

    @Override
    protected EntityManagerFactory createNativeEntityManagerFactory() throws
PersistenceException
    {
        ClassLoader original = getBeanClassLoader(); // save for later
        try
        {
            if (bundleContext != null)
            {
                // default
                String name = getPersistenceProviderClassName();
                PersistenceUnitInfo info = getPersistenceUnitInfo();
                if (info != null && !StringUtils.isEmpty(info
.getPersistenceProviderClassName()))
                {
                    // use class name of PU
                    name = info.getPersistenceProviderClassName();
                }

                if (StringUtils.isEmpty(getJkaApiBundleSymbolicName()))
                {
                    throw new IllegalStateException("no DataNucleus Jakarta API bundle
symbolic name given");
                }

                // set the bean class loader to use it so that Spring can find the
```

```
persistence provider class
                setBeanClassLoader(getBundleClassLoader(getJpaApiBundleSymbolicName(),
name));

                // since we're in an OSGi environment by virtue of the use of this
class, ensure a plugin registration mechanism is being used
                if (info == null || (info.getProperties() != null && !info
.getProperties().containsKey(getOsgiPluginRegistrarPropertyName())))
                {
                    Map<String, Object> map = getJpaPropertyMap();
                    map = map == null ? new HashMap<String, Object>() : map;
                    if (map.get(getOsgiPluginRegistrarPropertyName()) == null) {
                        map.put(getOsgiPluginRegistrarPropertyName(),
getOsgiPluginRegistrarClassName());
                    }
                }
            }

            // now let Springy do its thingy
            return super.createNativeEntityManagerFactory();
        }
        finally
        {
            setBeanClassLoader(original); // revert bean classloader
        }
    }

    protected ClassLoader getBundleClassLoader(String bundleSymbolicName,String
classNameToLoad)
    {
        ClassLoader classloader = null;
        Bundle[] bundles = bundleContext.getBundles();
        for (int x = 0; x < bundles.length; x++)
        {
            if (bundleSymbolicName.equals(bundles[x].getSymbolicName())) {
                try
                {
                    classloader = bundles[x].loadClass(classNameToLoad).
getClassLoader();
                }
                catch (ClassNotFoundException e)
                {
                    e.printStackTrace();
                }
                break;
            }
        }
        return classloader;
    }

    public String getJkaApiBundleSymbolicName() {
```

```
        return jkaApiBundleSymbolicName;
    }

    public void setJkaApiBundleSymbolicName(String jkaApiBundleSymbolicName) {
        this.jkaApiBundleSymbolicName = jkaApiBundleSymbolicName;
    }

    public String getPersistenceProviderClassName() {
        return persistenceProviderClassName;
    }

    public void setPersistenceProviderClassName(String persistenceProviderClassName) {
        this.persistenceProviderClassName = persistenceProviderClassName;
    }

    public String getOsgiPluginRegistrarClassName() {
        return osgiPluginRegistrarClassName;
    }

    public void setOsgiPluginRegistrarClassName(String osgiPluginRegistrarClassName) {
        this.osgiPluginRegistrarClassName = osgiPluginRegistrarClassName;
    }

    public String getOsgiPluginRegistrarPropertyName() {
        return osgiPluginRegistrarPropertyName;
    }

    public void setOsgiPluginRegistrarPropertyName(String
osgiPluginRegistrarPropertyName) {
        this.osgiPluginRegistrarPropertyName = osgiPluginRegistrarPropertyName;
    }
}
```

Performance Tuning

DataNucleus, by default, provides certain functionality. In particular circumstances some of this functionality may not be appropriate and it may be desirable to turn on or off particular features to gain more performance for the application in question. This section contains a few common tips

Enhancement

You should perform enhancement **before** runtime. That is, do not use *java agent* since it will enhance classes at runtime, when you want responsiveness from your application.

Schema

Jakarta Persistence provides properties for generating the schema at startup, and DataNucleus also provides some of its own (**datanucleus.schema.autoCreateAll**, **datanucleus.schema.autoCreateTables**, **datanucleus.schema.autoCreateColumns**, and **datanucleus.schema.autoCreateConstraints**). This can cause performance issues at startup. We recommend setting these to *false* at runtime, and instead using SchemaTool to **generate any required database schema before running DataNucleus (for RDBMS, HBase, etc)**.

Where you have an inheritance tree it is best to add a **discriminator** to the base class so that it's simple for DataNucleus to determine the class name for a particular row. For RDBMS : this results in cleaner/simpler SQL which is faster to execute, otherwise it would be necessary to do a UNION of all possible tables. For other datastores, a discriminator stores the key information necessary to instantiate the resultant class on retrieval so ought to be more efficient also.

DataNucleus provides 3 persistence properties (**datanucleus.schema.validateTables**, **datanucleus.schema.validateConstraints**, **datanucleus.schema.validateColumns**) that enforce strict validation of the datastore tables against the Meta-Data defined tables. This can cause performance issues at startup. In general this should be run only at schema generation, and should be turned off for production usage. Set all of these properties to *false*. In addition there is a property **datanucleus.rdbms.CheckExistTablesOrViews** which checks whether the tables/views that the classes map onto are present in the datastore. This should be set to *false* if you require fast start-up. Finally, the property **datanucleus.rdbms.initializeColumnInfo** determines whether the default values for columns are loaded from the database. This property should be set to *NONE* to avoid loading database metadata.

To sum up, the optimal settings with schema creation and validation disabled are:

```
#schema creation
datanucleus.schema.autoCreateAll=false
datanucleus.schema.autoCreateTables=false
datanucleus.schema.autoCreateColumns=false
datanucleus.schema.autoCreateConstraints=false

#schema validation
datanucleus.schema.validateTables=false
datanucleus.schema.validateConstraints=false
datanucleus.schema.validateColumns=false
datanucleus.rdbms.CheckExistTablesOrViews=false
datanucleus.rdbms.initializeColumnInfo=None
```

EntityManagerFactory usage

Creation of EntityManagerFactory objects can be expensive and should be kept to a minimum. Depending on the structure of your application, use a single factory per datastore wherever possible. Clearly if your application spans multiple servers then this may be impractical, but should be borne in mind.

You can improve startup speed by not specifying all classes in the *persistence-unit* so that they are discovered at runtime. Obviously this may impact on persistence operations later if classes are not known about.

Some RDBMS (such as Oracle) have trouble returning information across multiple catalogs/schemas and so, when DataNucleus starts up and tries to obtain information about the existing tables, it can take some time. This is easily remedied by specifying the catalog/schema name to be used - either for the EMF as a whole (using the persistence properties **datanucleus.Catalog**, **datanucleus.Schema**, or using the settings in `persistence.xml`), or for the package/class using attributes in the MetaData. This subsequently reduces the amount of information that the RDBMS needs to search through and so can give significant speed ups when you have many catalogs/schemas being managed by the RDBMS.

If you want to ensure that the schema existence checks are done for all persistence-unit classes at startup you should set the persistence property **datanucleus.persistenceUnitLoadClasses** to *true*. This processes all classes up front, meaning that all operations from there on will run faster without interruptions while it checks the database for existence of a table of a class.

EntityManager usage

Clearly the structure of your application will have a major influence on how you utilise an EntityManager. A pattern that gives a clean definition of process is to use a different persistence manager for each request to the data access layer. This reduces the risk of conflicts where one thread performs an operation and this impacts on the successful completion of an operation being performed by another thread. Creation of EM's is not an expensive process and use of multiple threads writing to the same manager should be avoided.

Make sure that you always close the EntityManager after use. It releases all resources connected to it, and failure to do so will result in memory leaks. Also note that when closing the EntityManager if you have the persistence property **datanucleus.detachOnClose** set to *true* (when in an **extended** PersistenceContext) this will detach all objects in the Level1 cache. Disable this if you don't need these objects to be detached, since it can be expensive when there are many objects.

Persistence Process

To optimise the persistence process for performance you need to analyse what operations are performed and when, to see if there are some features that you could disable to get the persistence you require and omit what is not required. If you think of a typical transaction, the following describes the process

- Start the transaction
- Perform persistence operations. If you are using "optimistic" transactions then all datastore operations will be delayed until commit. Otherwise all datastore operations will default to being performed immediately. If you are handling a very large number of objects in the transaction you would benefit by either disabling "optimistic" transactions, or alternatively setting the persistence property **datanucleus.flush.mode** to *AUTO*, or alternatively, do a manual flush every "n" objects, like this

```
for (int i=0;i<1000000;i++)
{
    if ((i%10000)/10000 == 0 && i != 0)
    {
        pm.flush();
    }
    ...
}
```

- Commit the transaction
  - All dirty objects are flushed.
  - Objects enlisted in the transaction are put in the Level 2 cache. You can disable the level 2 cache with the persistence property **datanucleus.cache.level2.type** set to *none*
  - Objects enlisted in the transaction are detached if you have the persistence property **datanucleus.detachAllOnCommit** set to *true* (when using a transactional PersistenceContext). Disable this if you don't need these objects to be detached at this point

Database Connection Pooling

DataNucleus, by default, will allocate connections when they are required. It then will close the connection.

In addition, when it needs to perform something via JDBC (RDBMS datastores) it will allocate a PreparedStatement, and then discard the statement after use. This can be inefficient relative to a database connection and statement pooling facility such as Apache DBCP. With Apache DBCP a

Connection is allocated when required and then when it is closed the Connection isn't actually closed but just saved in a pool for the next request that comes in for a Connection. This saves the time taken to establish a Connection and hence can give performance speed ups the order of maybe 30% or more. You can read about how to enable connection pooling with DataNucleus in the Connection Pooling Guide.

As an addendum to the above, you could also turn on caching of PreparedStatements. This can also give a performance boost, depending on your persistence code, the JDBC driver and the SQL being issued. Look at the persistence property **datanucleus.connectionPool.maxStatements**.

Retrieval of object by identity

If you are retrieving an object by its identity and know that it will be present in the Level2 cache, for example, you can set the persistence property **datanucleus.findObject.validateWhenCached** to *false* and this will skip a separate call to the datastore to validate that the object exists in the datastore.

Value Generators

DataNucleus provides a series of value generators for generation of identity values. These can have an impact on the performance depending on the choice of generator, and also on the configuration of the generator.

- The *max* strategy should not really be used for production since it makes a separate DB call for each insertion of an object. Something like the *TABLE* strategy should be used instead. Better still would be to choose *AUTO* and let DataNucleus decide for you.
- The *SEQUENCE* strategy allows configuration of the datastore sequence. The default can be non-optimum. As a guide, you can try setting **key-cache-size** to 10

The **AUTO** identity generator value is the recommended choice since this will allow DataNucleus to decide which identity generator is best for the datastore in use.

Collection/Map caching

DataNucleus has 2 ways of handling calls to SCO Collections/Maps. The original method was to pass all calls through to the datastore. The second method (which is now the default) is to cache the collection/map elements/keys/values. This second method will read the elements/keys/values once only and thereafter use the internally cached values. This second method gives significant performance gains relative to the original method. You can configure the handling of collections/maps as follows :-

- **Globally for the EMF** - this is controlled by setting the persistence property **datanucleus.cache.collections**. Set it to *true* for caching the collections (default), and *false* to pass through to the datastore.

- **For the specific Collection/Map** - this overrides the global setting and is controlled by adding a MetaData <*collection*> or <*map*> extension **cache**. Set it to *true* to cache the collection data, and *false* to pass through to the datastore.

The second method also allows a finer degree of control. This allows the use of lazy loading of data, hence elements will only be loaded if they are needed. You can configure this as follows :-

- **Globally for the EMF** - this is controlled by setting the property **datanucleus.cache.collections.lazy**. Set it to true to use lazy loading, and set it to false to load the elements when the collection/map is initialised.
- **For the specific Collection/Map** - this overrides the global EMF setting and is controlled by adding a MetaData <*collection*> or <*map*> extension **cache-lazy-loading**. Set it to *true* to use lazy loading, and *false* to load once at initialisation.

NonTransactional Reads (Reading persistent objects outside a transaction)

Performing non-transactional reads has advantages and disadvantages in performance and data freshness in cache. The objects read are held cached by the EntityManager. The second time an application requests the same objects from the EntityManager they are retrieved from cache. The time spent reading the object from cache is minimum, but the objects may become stale and not represent the database status. If fresh values need to be loaded from the database, then the user application should first call *refresh* on the object.

Another disadvantage of performing non-transactional reads is that each operation realized opens a new database connection, but it can be minimized with the use of connection pools, and also on some of the datastore the (nontransactional) connection is retained.

Accessing fields of persistent objects when not managed by a EntityManager

Reading fields of unmanaged objects (outside the scope of an *EntityManager*) is a trivial task, but performed in a certain manner can determine the application performance. The objective here is not give you an absolute response on the subject, but point out the benefits and drawbacks for the many possible solutions.

- Use **datanucleus.RetainValues**=true. This is the default for Jakarta Persistence operation and will ensure that after commit the fields of the object retain their values (rather than being nulled).
- Use *detach* method.

```
Object copy = null;
try
{
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    //retrieve in some way the object, query, find, etc
    Object obj = em.find(MyClass.class, id);
    copy = em.detach(obj);

    em.getTransaction().commit();
}
finally
{
    em.close();
}
//read or change the detached object here
System.out.prinln(copy.getName());
```

- Use **datanucleus.detachAllOnCommit**=true. Dependent on the persistence context you may automatically have this set.

```
Object obj = null;
try
{
    EntityManager pm = emf.createEntityManager();
    em.getTransaction().begin();

    //retrieve in some way the object, query, find, etc
    obj = em.find(MyClass.class, id);
    em.getTransaction().commit(); // Object "obj" is now detached
}
finally
{
    em.close();
}
//read or change the detached object here
System.out.prinln(obj.getName());
```

The bottom line is to not use detachment if instances will only be used to read values.

Fetch Control

When fetching objects you have control over what gets fetched. This can have an impact if you are then detaching those objects. With Jakarta Persistence the maximum fetch depth is -1 (unlimited). So with Jakarta Persistence you ought to set it to the extent that you want to detach, or better still

make use of Jakarta Entity Graphs to control the specific fields to detach.

Logging

I/O consumes a huge slice of the total processing time. Therefore it is recommended to reduce or disable logging in production. To disable the logging set the DataNucleus category to OFF in the Log4j (v1) configuration. See Logging for more information.

```
log4j.category.DataNucleus=OFF
```

General Comments

In most applications, the performance of the persistence layer is very unlikely to be a bottleneck. More likely the design of the datastore itself, and in particular its indices are more likely to have the most impact, or alternatively network latency. That said, it is the DataNucleus projects' committed aim to provide the best performance possible, though we also want to provide functionality, so there is a compromise with respect to resource.

A benchmark is defined as "a series of persistence operations performing particular things e.g persist *n* objects, or retrieve *n* objects". If those operations are representative of your application then the benchmark is valid to you.

To find (or create) a benchmark appropriate to your project you need to determine the typical persistence operations that your application will perform. Are you interested in persisting 100 objects at once, or 1 million, for example? Then when you have a benchmark appropriate for that operation, compare the persistence solutions.

The performance tuning guide above gives a good oversight of tuning capabilities, and also refer to the following blog entry for our take on performance of DataNucleus AccessPlatform. And then the later blog entry about how to tune for bulk operations

Object-NoSQL Database Mappers: a benchmark study on the performance overhead (Dec 2016)

This paper makes an attempt to compare several mappers for MongoDB, comparing with native MongoDB usage. Key points to make are

- The study persists a flat class, with no relations. Hardly representative of a real world usage.
- The study doesn't even touch on feature set available in each mapper, so the fact that DataNucleus has a very wide range of mapping capabilities for MongoDB is ignored.
- All mappers come out as slower than native MongoDB (surprise!). The whole point of using a mapper is that you don't want to spend the time learning a new API, so are prepared for some overhead.
- All timings quoted in their report are in the "microseconds" range!! as are differences between the methods so very few real world applications would be impacted by the differences shown. If anybody is choosing a persistence mechanism for pure speed, they should **always** go with the

native API; right tool for the job.

- DataNucleus was configured to turn OFF query compilation caching, and L2 caching !!! whereas not all other mappers provide a way to not cache such things, hence they have tied one arm behind its back, and then commented that time taken to compile queries is impacting on performance!
- Enhancement was done at RUNTIME!! so would impact on performance results. Not sure how many times we need to say this in reference to benchmarking but clearly the message hasn't got through, or to quote the report "*this may indicate fundamental flaws in the study's measurement methodology*".
- This uses v5.0.0.M5. Not sure why each benchmark we come across wants to use some milestone (used for DataNucleus) rather than a full release (what they did for all other mappers). There have been changes to core performance since early 5.0

GeeCon JPA provider comparison (Jun 2012)

There is an interesting presentation on JPA provider performance that was presented at GeeCon 2012 by Patrycja Wegrzynowicz. This presentation takes the time to look at what operations the persistence provider is performing, and does more than just "persist large number of flat objects into a single table", and so gives you something more interesting to analyse. DataNucleus comes out pretty well in many situations. You can also see the PDF here.

PolePosition (Dec 2008)

The PolePosition benchmark is a project on SourceForge to provide a benchmark of the write, read and delete of different data structures using the various persistence tools on the market. JPOX (DataNucleus predecessor) was run against this benchmark just before being renamed as DataNucleus and the following conclusions about the benchmark were made.

- It is essential that tests for such as Hibernate and DataNucleus performance comparable things. Some of the original tests had the "delete" simply doing a "DELETE FROM TBL" for Hibernate yet doing an Extent followed by delete each object individually for a JDO implementation. This is an unfair comparison and in the source tree in JPOX SVN this is corrected. This fix was pointed out to the PolePos SourceForge project but is not, as yet, fixed
- It is essential that schema is generated before the test, otherwise the test is no longer a benchmark of just a persistence operation. The source tree in JPOX SVN assumes the schema exists. This fix was pointed out to the PolePos SourceForge project but is not, as yet, fixed
- Each persistence implementation should have its own tuning options, and be able to add things like discriminators since that is what would happen in a real application. The source tree in JPOX SVN does this for JPOX running. Similarly a JDO implementation would tune the entity graphs being used - this is not present in the SourceForge project but is in JPOX SVN.
- DataNucleus performance is considered to be significantly improved over JPOX particularly due to batched inserts, and due to a rewritten query implementation that does enhanced fetching.

Replication

Many applications make use of multiple datastores. It is a common requirement to be able to replicate parts of one datastore in another datastore. Obviously, depending on the datastore, you could make use of the datastores own capabilities for replication. DataNucleus provides its own extension to Jakarta Persistence to allow replication from one datastore to another. This extension doesn't restrict you to using 2 datastores of the same type. You could replicate from RDBMS to XML for example, or from MySQL to HSQLDB.

You need to make sure you have the persistence property *datanucleus.attachSameDatastore* set to *false* if using replication

Note that the case of replication between two RDBMS of the same type is usually way more efficiently replicated using the capabilities of the datastore itself

The following sample code will replicate all objects of type *Product* and *Employee* from EMF1 to EMF2. These EMFs are created in the normal way so, as mentioned above, EMF1 could be for a MySQL datastore, and EMF2 for XML. By default this will replicate the complete object graphs reachable from these specified types.

```
import org.datanucleus.api.jakarta.JakartaReplicationManager;

...

JakartaReplicationManager replicator = new JakartaReplicationManager(emf1, emf2);
replicator.replicate(new Class[]{Product.class, Employee.class});
```

Monitoring

DataNucleus allows a user to enable various MBeans internally. These can then be used for monitoring the number of datastore calls etc.

Via API

The simplest way to monitor DataNucleus is to use its API for monitoring. Internally there are several MBeans (as used by JMX) and you can navigate to these to get the required information. To enable this set the persistence property **datanucleus.enableStatistics** to *true*. There are then two sets of statistics; one for the EMF and one for each EM. You access these as follows

```
JakartaEntityManagerFactory dnEMF = (JakartaEntityManagerFactory)emf;
FactoryStatistics stats = dnEMF.getNucleusContext().getStatistics();
... (access the statistics information)


JakartaEntityManager dnEM = (JakartaEntityManager)em;
ManagerStatistics stats = dnEM.getExecutionContext().getStatistics();
... (access the statistics information)
```

Using JMX

The MBeans used by DataNucleus can be accessed via JMX at runtime. More about JMX here.

An MBean server is bundled with Sun/Oracle JRE since Java5, and you can easily activate DataNucleus MBeans registration by creating your EMF with the persistence property **datanucleus.jmxType** as *platform*

Additionally, setting a few system properties are necessary for configuring the Sun JMX implementation. The minimum properties required are the following:

- com.sun.management.jmxremote
- com.sun.management.jmxremote.authenticate
- com.sun.management.jmxremote.ssl
- com.sun.management.jmxremote.port=<port number>

Usage example:

```
java -cp TheClassPathInHere
     -Dcom.sun.management.jmxremote
     -Dcom.sun.management.jmxremote.authenticate=false
     -Dcom.sun.management.jmxremote.ssl=false
     -Dcom.sun.management.jmxremote.port=8001
     TheMainClassInHere
```

Once you start your application and DataNucleus is initialized you can browse DataNucleus MBeans using a tool called jconsole (jconsole is distributed with the Sun JDK) via the URL:

```
service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi
```

Note that the mode of usage is presented in this document as matter of example, and by no means we recommend to disable authentication and secured communication channels. Further details on the Sun JMX implementation and how to configure it properly can be found here.

DataNucleus MBeans are registered in a MBean Server when DataNucleus is started up (e.g. upon Jakarta EMF instantiation). To see the full list of DataNucleus MBeans, refer to the javadocs.

DataNucleus Logging (v6.0)

DataNucleus can be configured to log significant amounts of information regarding its process. This information can be very useful in tracking the persistence process, and particularly if you have problems. DataNucleus will log as follows :-

- **Log4J v2** - if you have Log4J v2 in the CLASSPATH, Apache Log4J v2 will be used
- **Log4J v1** - otherwise if you have Log4J v1 in the CLASSPATH, Apache Log4J v1 will be used
- **java.util.logging** - otherwise if you don't have Log4J in the CLASSPATH, then *java.util.logging* will be used

DataNucleus logs messages to various categories (in Log4J and java.util.logging these correspond to a "Logger"), allowing you to filter the logged messages by these categories - so if you are only interested in a particular category you can effectively turn the others off. DataNucleus's log is written by default in English. If your JRE is running in a Spanish locale then your log will be written in Spanish.

If you have time to translate our log messages into other languages, please contact one of the developers via Groups.IO

Logging Categories

DataNucleus uses a series of **categories**, and logs all messages to these **categories**. Currently DataNucleus uses the following

- **DataNucleus.Persistence** - All messages relating to the persistence process
- **DataNucleus.Transaction** - All messages relating to transactions
- **DataNucleus.Connection** - All messages relating to Connections.
- **DataNucleus.Query** - All messages relating to queries
- **DataNucleus.Cache** - All messages relating to the DataNucleus Cache
- **DataNucleus.MetaData** - All messages relating to MetaData
- **DataNucleus.Datastore** - All general datastore messages
- **DataNucleus.Datastore.Schema** - All schema related datastore log messages
- **DataNucleus.Datastore.Persist** - All datastore persistence messages
- **DataNucleus.Datastore.Retrieve** - All datastore retrieval messages
- **DataNucleus.Datastore.Native** - Log of all 'native' statements sent to the datastore
- **DataNucleus.General** - All general operational messages
- **DataNucleus.Lifecycle** - All messages relating to object lifecycle changes
- **DataNucleus.ValueGeneration** - All messages relating to value generation
- **DataNucleus.Enhancer** - All messages from the DataNucleus Enhancer.
- **DataNucleus.SchemaTool** - All messages from DataNucleus SchemaTool

- **DataNucleus.JDO** - All messages general to JDO
- **DataNucleus.JPA** - All messages general to JPA
- **DataNucleus.JCA** - All messages relating to Connector JCA.
- **DataNucleus.IDE** - Messages from the DataNucleus IDE.

Using Log4J v2

Log4J allows logging messages at various severity levels. The levels used by Log4J, and by DataNucleus's use of Log4J are **DEBUG**, **INFO**, **WARN**, **ERROR**, **FATAL**. Each message is logged at a particular level to a **category** (as described above). The other setting is **OFF** which turns off a logging category; very useful in a production situation where maximum performance is required.

To enable the DataNucleus log, you need to provide a Log4J configuration file when starting up your application. This may be done for you if you are running within a JavaEE application server (check your manual for details). If you are starting your application yourself, you would set a JVM parameter as

```
-Dlog4j.configurationFile=file:log4j2.xml
```

where `log4j2.xml` is the name of your Log4J v2 configuration file. Please note the *file:* prefix to the file since a URL is expected.

The Log4J configuration file is very simple in nature, and you typically define where the log goes to (e.g to a file), and which logging level messages you want to see. Here's an example

```
<Configuration status="info" strict="true" name="datanucleus">

    <Properties>
        <Property name="dnFilename">datanucleus.log</Property>
    </Properties>

    <Appenders>
        <Appender type="File" name="DataNucleus" fileName="${dnFilename}">
            <Layout type="PatternLayout" pattern="%d{HH:mm:ss,SSS} (%t) %-5p [%c] -
%m%n"/>
        </Appender>
    </Appenders>

    <Loggers>
        <!-- DataNucleus Loggers (all) -->
        <Logger name="DataNucleus" level="warn" additivity="false">
            <AppenderRef ref="DataNucleus"/>
        </Logger>

        <Root level="error">
            <AppenderRef ref="DataNucleus"/>
        </Root>
    </Loggers>
```

In this example, I am directing my log to a file (`datanucleus.log`). I have defined a particular "pattern" for the messages that appear in the log (to contain the date, level, category, and the message itself). You could configure each Logger at a different level. e.g "DataNucleus.MetaData" could be at level *debug*.

Turning OFF the logging, or at least down to ERROR level provides a *significant* improvement in performance. With Log4J v2 you set the *level* to **OFF**.

Using Log4J v1

The same applies as for Log4j v2.

To enable the DataNucleus log, you need to provide a Log4J configuration file when starting up your application. This may be done for you if you are running within a JavaEE application server (check your manual for details). If you are starting your application yourself, you would set a JVM parameter as

```
-Dlog4j.configuration=file:log4j.properties
```

where `log4j.properties` is the name of your Log4J configuration file. Please note the *file:* prefix to the file since a URL is expected.

The Log4J configuration file is very simple in nature, and you typically define where the log goes to

(e.g to a file), and which logging level messages you want to see. Here's an example

```
# Define the destination and format of our logging
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.File=datanucleus.log
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{HH:mm:ss,SSS} (%t) %-5p [%c] - %m%n

# DataNucleus Categories
log4j.category.DataNucleus.JDO=INFO, A1
log4j.category.DataNucleus.Cache=INFO, A1
log4j.category.DataNucleus.MetaData=INFO, A1
log4j.category.DataNucleus.General=INFO, A1
log4j.category.DataNucleus.Transaction=INFO, A1
log4j.category.DataNucleus.Datastore=DEBUG, A1
log4j.category.DataNucleus.ValueGeneration=DEBUG, A1

log4j.category.DataNucleus.Enhancer=INFO, A1
log4j.category.DataNucleus.SchemaTool=INFO, A1
```

In this example, I am directing my log to a file (`datanucleus.log`). I have defined a particular "pattern" for the messages that appear in the log (to contain the date, level, category, and the message itself). In addition I have assigned a level "threshold" for each of the DataNucleus **categories**. So in this case I want to see all messages down to DEBUG level for the DataNucleus RDBMS persister.

Turning OFF the logging, or at least down to ERROR level provides a *significant* improvement in performance. With Log4J you do this via

```
log4j.category.DataNucleus=OFF
```

Using java.util.logging

java.util.logging allows logging messages at various severity levels. The levels used by java.util.logging, and by DataNucleus's internally are **fine**, **info**, **warn**, **severe**. Each message is logged at a particular level to a **category** (as described above).

By default, the *java.util.logging* configuration is taken from a properties file `<JRE_DIRECTORY>/lib/logging.properties`. Modify this file and configure the categories to be logged, or use the **java.util.logging.config.file** system property to specify a properties file (in java.util.Properties format) where the logging configuration will be read from. Here is an example:

```
handlers=java.util.logging.FileHandler, java.util.logging.ConsoleHandler
DataNucleus.General.level=fine
DataNucleus.JDO.level=fine

# --- ConsoleHandler ---
# Override of global logging level
java.util.logging.ConsoleHandler.level=SEVERE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

# --- FileHandler ---
# Override of global logging level
java.util.logging.FileHandler.level=SEVERE

# Naming style for the output file:
java.util.logging.FileHandler.pattern=datanucleus.log

# Limiting size of output file in bytes:
java.util.logging.FileHandler.limit=50000

# Number of output files to cycle through, by appending an
# integer to the base file name:
java.util.logging.FileHandler.count=1

# Style of output (Simple or XML):
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

Please read the javadocs for *java.util.logging* for additional details on its configuration.

Sample Log Output

Here is a sample of the type of information you may see in the DataNucleus log when using Log4J.

```
21:26:09,000 (main) INFO  DataNucleus.Datastore.Schema - Adapter initialised :
MySQLAdapter, MySQL version 4.0.11
21:26:09,365 (main) INFO  DataNucleus.Datastore.Schema - Creating table
null.DELETE_ME1080077169045
21:26:09,370 (main) DEBUG DataNucleus.Datastore.Schema - CREATE TABLE
DELETE_ME1080077169045
(
    UNUSED INTEGER NOT NULL
) TYPE=INNODB
21:26:09,375 (main) DEBUG DataNucleus.Datastore.Schema - Execution Time = 3 ms
21:26:09,388 (main) WARN  DataNucleus.Datastore.Schema - Schema Name could not be
determined for this datastore
21:26:09,388 (main) INFO  DataNucleus.Datastore.Schema - Dropping table
null.DELETE_ME1080077169045
21:26:09,388 (main) DEBUG DataNucleus.Datastore.Schema - DROP TABLE
DELETE_ME1080077169045
21:26:09,392 (main) DEBUG DataNucleus.Datastore.Schema - Execution Time = 3 ms
21:26:09,392 (main) INFO  DataNucleus.Datastore.Schema - Initialising Schema "" using
"SchemaTable" auto-start
21:26:09,401 (main) DEBUG DataNucleus.Datastore.Schema - Retrieving type for table
DataNucleus_TABLES
21:26:09,406 (main) INFO  DataNucleus.Datastore.Schema - Creating table
null.DataNucleus_TABLES
21:26:09,406 (main) DEBUG DataNucleus.Datastore.Schema - CREATE TABLE
DataNucleus_TABLES
(
    CLASS_NAME VARCHAR (128) NOT NULL UNIQUE ,
    `TABLE_NAME` VARCHAR (127) NOT NULL UNIQUE
) TYPE=INNODB
21:26:09,416 (main) DEBUG DataNucleus.Datastore.Schema - Execution Time = 10 ms
21:26:09,417 (main) DEBUG DataNucleus.Datastore - Retrieving type for table
DataNucleus_TABLES
21:26:09,418 (main) DEBUG DataNucleus.Datastore - Validating table :
null.DataNucleus_TABLES
21:26:09,425 (main) DEBUG DataNucleus.Datastore - Execution Time = 7 ms
```

So you see the time of the log message, the level of the message (DEBUG, INFO, etc), the category (DataNucleus.Datastore, etc), and the message itself. For example, if I had set the *DataNucleus.Datastore.Schema* to DEBUG and all other categories to INFO I would see **all** DDL statements sent to the database and very little else.

HOWTO : Log with log4j and DataNucleus under OSGi

This guide was provided by Marco Lopes, when using DataNucleus v2.2. All of the bundles which use log4j should have *org.apache.log4j* in their Import-Package attribute! (use: *org.apache.log4j;resolution:=optional* if you don't want to be stuck with log4j whenever you use an edited bundle in your project!).

Method 1

- Create a new "Fragment Project". Call it whatever you want (ex: log4j-fragment)
- You have to define a "Plugin-ID", that's the plugin where DN will run
- Edit the MANIFEST
- Under RUNTIME add log4j JAR to the Classpath
- Under Export-Packages add org.apache.log4j
- Save MANIFEST
- PASTE the log4j PROPERTIES file into the SRC FOLDER of the Project

Method 2

- Get an "OSGI Compliant" log4j bundle (you can get it from the SpringSource Enterprise Bundle Repository
- Open the Bundle JAR with WINRAR (others might work)
- PASTE the log4j PROPERTIES file into the ROOT of the bundle
- Exit. Winrar will ask to UPDATE the JAR. Say YES.
- Add the updated OSGI compliant Log4j bundle to your Plugin Project Dependencies (Required-Plugins)

Each method has it's own advantages. Use method 1 if you need to EDIT the log4j properties file ON-THE-RUN. The disadvantage: it can only "target" one project at a time (but very easy to edit the MANIFEST and select a new Host Plugin!). Use method 2 if you want to have log4j support in every project with only one file. The disadvantage: it's not very practical to edit the log4j PROPERTIES file (not because of the bundle EDIT, but because you have to restart eclipse in order for the new bundle to be recognized).