JDO Persistence Guide (v5.1)

Table of Contents

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

You should familiarise yourself with the JDO 3.2 Javadocs.

PersistenceManagerFactory

Any JDO-enabled application will require at least one *PersistenceManagerFactory* (PMF) [Javadoc]. Typically applications create one per datastore being utilised. A *PersistenceManagerFactory* provides access to *PersistenceManager(s)* which allow objects to be persisted, and retrieved. The *PersistenceManagerFactory* can be configured to provide particular behaviour.

> A *PersistenceManagerFactory* is designed to be thread-safe. A *PersistenceManager* is not.

> A *PersistenceyManagerFactory* 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 *PersistenceManagerFactory* after you have finished with it.

There are many ways of creating a *PersistenceManagerFactory*, some of which are shown below

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
"org.datanucleus.api.jdo.JDOPersistenceManagerFactory");
properties.setProperty("javax.jdo.option.ConnectionURL","jdbc:mysql://localhost/myDB")
;
properties.setProperty("javax.jdo.option.ConnectionUserName","login");
properties.setProperty("javax.jdo.option.ConnectionPassword","password");

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

A slight variation on this, is to have a file to specify these properties in a file

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.api.jdo.JDOPersistenceManager
Factory
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/myDB
javax.jdo.option.ConnectionUserName=login
javax.jdo.option.ConnectionPassword=password
```

and then to create the *PersistenceManagerFactory* using this file

```
File propsFile = new File(filename);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(propsFile);
```

or if the above file is in the CLASSPATH (at `datanucleus.properties` in the root of the CLASSPATH), then

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory
("datanucleus.properties");
```

If using a *named PMF* file, you can create the PMF by providing the [name of the PMF](#) like this

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("myNamedPMF");
```

If using a `META-INF/persistence.xml` file, you can simply specify the [persistence-unit](#) name as

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory
("myPersistenceUnit");
```

Another alternative, when specifying your datastore via JNDI, would be to call *JDOHelper.getPersistenceManagerFactory(jndiLocation, context);*, and then set the other persistence properties on the received PMF.

Whichever way we wish to obtain the *PersistenceManagerFactory* we have defined a series of properties to give the behaviour of the *PersistenceManagerFactory*. The first property specifies to use the DataNucleus implementation, and the following 4 properties define the datastore that it should connect to. There are many properties available. Some of these are standard JDO properties, and some are DataNucleus extensions.

PersistenceManagerFactory for Persistence-Unit

When designing an application you can usually nicely separate your persistable objects into independent groupings that can be treated separately, perhaps within a different DAO object, if using DAOs. JDO uses the (JPA) idea of a *persistence-unit*. A *persistence-unit* provides a convenient way of specifying a set of metadata files, and classes, and jars that contain all classes to be persisted in a grouping. The persistence-unit is named, and the name is used for identifying it. Consequently this name can 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 the CLASSPATH (this may mean `WEB-INF/classes/META-INF` when using a web-application in such as Tomcat). This file will be used to define your *persistence-unit(s)*. Lets show an example

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persistence 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_1.xsd" version="2.1">

    <!-- Online Store -->
    <persistence-unit name="OnlineStore">
        <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="datanucleus.ConnectionURL" value=
"jdbc:h2:mem:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" value=""/>
        </properties>
    </persistence-unit>

    <!-- Accounting -->
    <persistence-unit name="Accounting">
        <mapping-file>/mydomain/samples/metadata/accounts/package.jdo</mapping-file>
        <properties>
            <property name="datanucleus.ConnectionURL" value=
"jdbc:h2:mem:datanucleus"/>
            <property name="datanucleus.ConnectionUserName" value="sa"/>
            <property name="datanucleus.ConnectionPassword" 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 `package.jdo` 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

- **provider** - Not used by JDO

- **jta-data-source** - JNDI name for JTA connections (make sure you set *transaction-type* as **JTA** on the persistence-unit for this). You can alternatively specify JDO standard **javax.jdo.option.ConnectionFactoryName** to the same end.

- **non-jta-data-source** - JNDI name for non-JTA connections. You can alternatively specify JDO

standard **javax.jdo.option.ConnectionFactory2Name** to the same end.

- **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. This is the JDO XML Metadata file (*package.jdo*) (**not** the ORM XML Metadata file)

- **class** - name of an annotated class to include in this persistence-unit

- **properties** - properties defining the persistence factory to be used.

- **exclude-unlisted-classes** - when this is specified then it will only load metadata for the classes/mapping files listed.

Use with JDO

JDO accepts the "persistence-unit" name to be specified when creating the *PersistenceManagerFactory*, like this

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory
("MyPersistenceUnit");
```

Metadata loading using persistence unit

When you specify a PMF using a `persistence.xml` it will load the metadata for all classes that are specified directly in the persistence unit, as well as all classes defined in JDO XML metadata files 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.

Dynamically generated Persistence-Unit

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

```
import org.datanucleus.metadata.PersistenceUnitMetaData;
import org.datanucleus.api.jdo.JDOPersistenceManagerFactory;

PersistenceUnitMetaData pumd = new PersistenceUnitMetaData("dynamic-unit",
"RESOURCE_LOCAL", null);
pumd.addClassName("mydomain.test.A");
pumd.setExcludeUnlistedClasses();
pumd.addProperty("javax.jdo.ConnectionURL", "jdbc:hsqldb:mem:nucleus");
pumd.addProperty("javax.jdo.ConnectionUserName", "sa");
pumd.addProperty("javax.jdo.ConnectionPassword", "");
pumd.addProperty("datanucleus.schema.autoCreateAll", "true");

PersistenceManagerFactory pmf = new JDOPersistenceManagerFactory(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.

Named PersistenceManagerFactory

Typically applications create one PMF per datastore being utilised. An alternate to persistence-unit is to use a **named PMF**, defined in a file `META-INF/jdoconfig.xml` at the root of the CLASSPATH (this may mean `WEB-INF/classes/META-INF` when using a web-application). Let's see an example of a `jdoconfig.xml`

```xml
<?xml version="1.0" encoding="utf-8"?>
<jdoconfig xmlns="http://xmlns.jcp.org/xml/ns/jdo/jdoconfig"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/jdoconfig
        http://xmlns.jcp.org/xml/ns/jdo/jdoconfig_3_2.xsd" version="3.2">

    <!-- Datastore Txn PMF -->
    <persistence-manager-factory name="Datastore">
        <property name="javax.jdo.PersistenceManagerFactoryClass"
value="org.datanucleus.api.jdo.JDOPersistenceManagerFactory"/>
        <property name="javax.jdo.option.ConnectionURL"
value="jdbc:mysql://localhost/datanucleus?useServerPrepStmts=false"/>
        <property name="javax.jdo.option.ConnectionUserName" value="datanucleus"/>
        <property name="javax.jdo.option.ConnectionPassword" value=""/>
        <property name="javax.jdo.option.Optimistic" value="false"/>
        <property name="datanucleus.schema.autoCreateAll" value="true"/>
    </persistence-manager-factory>

    <!-- Optimistic Txn PMF -->
    <persistence-manager-factory name="Optimistic">
        <property name="javax.jdo.PersistenceManagerFactoryClass"
value="org.datanucleus.api.jdo.JDOPersistenceManagerFactory"/>
        <property name="javax.jdo.option.ConnectionURL"
value="jdbc:mysql://localhost/datanucleus?useServerPrepStmts=false"/>
        <property name="javax.jdo.option.ConnectionUserName" value="datanucleus"/>
        <property name="javax.jdo.option.ConnectionPassword" value=""/>
        <property name="javax.jdo.option.Optimistic" value="true"/>
        <property name="datanucleus.schema.autoCreateAll" value="true"/>
    </persistence-manager-factory>

</jdoconfig>
```

So in this example we have 2 named PMFs. The first is known by the name "Datastore" and utilises datastore transactions. The second is known by the name "Optimistic" and utilises optimistic locking. You simply define all properties for the particular PMF within its specification block. And finally we instantiate our PMF like this

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Optimistic");
```

That's it. The PMF we are returned from JDOHelper will have all of the properties defined in META-INF/jdoconfig.xml under the name of "Optimistic".

PersistenceManagerFactory Properties

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

Specifying the datastore properties

With JDO you have 3 ways of specifying the datastore via persistence properties

- **Specify the connection URL/userName/password(/driverName)** and it will internally create a DataSource for this URL (with optional connection pooling). This is achieved by specifying **javax.jdo.option.ConnectionURL**, **javax.jdo.option.ConnectionUserName**, **javax.jdo.option.ConnectionPassword** and optionally **javax.jdo.option.ConnectionDriverName**.

- **Specify the JNDI name of the connectionFactory**. This is achieved by specifying **javax.jdo.option.ConnectionFactoryName**, and **javax.jdo.option.ConnectionFactory2Name** (for secondary operations)

- **Specify the DataSource of the connectionFactory**. This is achieved by specifying **javax.jdo.option.ConnectionFactory**, and **javax.jdo.option.ConnectionFactory2** (for secondary operations)

The JNDI routes are typically only for use with RDBMS datastores.

> The "ConnectionURL" value for the different supported datastores is defined in the Datastore Guide

Standard JDO Properties

Parameter	Description + Values
javax.jdo.PersistenceManagerFactoryClass	The name of the PMF implementation. *org.datanucleus.api.jdo.JDOPersistenceManagerFactory* **Only required if you have more than one JDO implementation in the CLASSPATH**
javax.jdo.option.ConnectionFactory	Instance of a connection factory for **transactional** connections. This is an alternative to specifying the ConnectionURL. **Only for RDBMS**, and it must be an instance of javax.sql.DataSource. See here
javax.jdo.option.ConnectionFactory2	Instance of a connection factory for **nontransactional** connections. This is an alternative to specifying the ConnectionURL. **Only for RDBMS**, and it must be an instance of javax.sql.DataSource. See here
javax.jdo.option.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 here
javax.jdo.option.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 here
javax.jdo.option.ConnectionURL	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.
javax.jdo.option.ConnectionUserName	Username to use for connecting to the DB

Parameter	Description + Values
javax.jdo.option.ConnectionPassword	Password to use for connecting to the DB. See **datanucleus.ConnectionPasswordDecrypter** for a way of providing an encrypted password here
javax.jdo.option.ConnectionDriverName	The name of the driver to use for the DB. **For RDBMS, and not needed for JDBC4+ drivers**. Note that some 3rd party connection pools do require the driver class name still. **For LDAP, specifying the initial context factory**.
javax.jdo.option.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**}
javax.jdo.option.Multithreaded	Whether to try to run the PM multithreaded. **Note that this is only a hint to try to allow thread-safe operations on the PM**. Users are always advised to run a PM as single threaded, since some operations are not currently locked and so could cause issues multi-threaded. {true, **false**}
javax.jdo.option.Optimistic	Whether to use optimistic locking. {true, **false**}
javax.jdo.option.RetainValues	Whether to suppress the clearing of values from persistent instances on transaction completion. {true, **false**}
javax.jdo.option.RestoreValues	Whether persistent object have transactional field values restored when transaction rollback occurs. {true, **false**}
javax.jdo.option.DetachAllOnCommit	Allows the user to select that when a transaction is committed all objects enlisted in that transaction will be automatically detached. {true, **false**}
javax.jdo.option.CopyOnAttach	Whether, when attaching a detached object, we create an attached copy or simply migrate the detached object to attached state {**true**, false}
javax.jdo.option.PersistenceUnitName	Name of a *persistence-unit* to be found in a `persistence.xml` file (under META-INF) that defines the persistence properties to use and the classes to use within the persistence process.
javax.jdo.option.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.
javax.jdo.option.Name	Name of the named PMF to use. Refers to a PMF defined in `META-INF/jdoconfig.xml`.
javax.jdo.option.ReadOnly	Whether the datastore is read-only or not (fixed in structure and contents). {true, **false**}
javax.jdo.option.TransactionType	Type of transaction to use. {RESOURCE_LOCAL, JTA}

Parameter	Description + Values
javax.jdo.option.TransactionIsolationLevel	Select the default transaction isolation level for ALL PM factories. 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}
javax.jdo.option.NontransactionalRead	Whether to allow nontransactional reads {false, **true**}
javax.jdo.option.NontransactionalWrite	Whether to allow nontransactional writes {false, **true**}
javax.jdo.option.DatastoreReadTimeoutMillis	The timeout to apply to all reads (millisecs), e.g by query or by PM.getObjectById(). **Only applies if the underlying datastore supports it {0**, A positive value (MILLISECONDS)}
javax.jdo.option.DatastoreWriteTimeoutMillis	The timeout to apply to all writes (millisecs). **Only applies if the underlying datastore supports it {0**, A positive value (MILLISECONDS)}
javax.jdo.option.Mapping	Name for the ORM MetaData mapping files to use with this PMF. For example if this is set to "mysql" then the implementation looks for MetaData mapping files called `{classname}-mysql.orm` or `package-mysql.orm`. If this is not specified then the JDO implementation assumes that all is specified in the JDO MetaData file.
javax.jdo.mapping.Catalog	Name of the catalog to use by default for all classes persisted using this PMF. 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.
javax.jdo.mapping.Schema	Name of the schema to use by default for all classes persisted using this PMF. 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.

DataNucleus Datastore Properties

Extension

DataNucleus provides the following properties for configuring the datastore used by the PersistenceManagerFactory.

Parameter	Description + Values
datanucleus.ConnectionURL	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.
datanucleus.ConnectionUserName	Username to use for connecting to the DB

Parameter	Description + Values
datanucleus.ConnectionPassword	Password to use for connecting to the DB. See property **datanucleus.ConnectionPasswordDecrypter** for a way of providing an encrypted password here
datanucleus.ConnectionDriverName	The name of the (JDBC) driver to use for the DB; **For RDBMS, defining the driver name, but not needed for JDBC 4+ drivers**, **For LDAP, specifying the initial context factory**.
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. 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. 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 Persistence Properties

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

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 run the PM multithreaded. **Note that this is only a hint to try to allow thread-safe operations on the PM**. Users are always advised to run a PM 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**}

Parameter	Description + Values
datanucleus.RestoreValues	Whether persistent object have transactional field values restored when transaction rollback occurs. {true, **false**}
datanucleus.Mapping	Name for the ORM MetaData mapping files to use with this PMF. For example if this is set to "mysql" then the implementation looks for MetaData mapping files called `{classname}-mysql.orm` or `package-mysql.orm`. If this is not specified then the JDO implementation assumes that all is specified in the JDO MetaData file.
datanucleus.mapping.Catalog	Name of the catalog to use by default for all classes persisted using this PMF. 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 datastores only*
datanucleus.mapping.Schema	Name of the schema to use by default for all classes persisted using this PMF. 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 datastores 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 datastores 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 datastores 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}

Parameter	Description + Values
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}
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 a PM 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. {**false**, true}
datanucleus.detachmentFields	When detaching you can control what happens to loaded/unloaded fields of the FetchPlan. The default for JDO 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 JDO spec defines a default of 1, meaning that only the first level of related objects will be fetched by default. {-1, **1**, positive integer (non-zero)}
datanucleus.detachedState	Allows control over which mechanism to use to determine the fields to be detached. By default DataNucleus uses the defined "fetch-groups". JPA 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.PersistenceUnitName	Name of a *persistence-unit* to be found in a `persistence.xml` file (under META-INF) that defines the persistence properties to use and the classes to use within the persistence process.

Parameter	Description + Values
datanucleus.PersistenceUnitLoadClasses	Used when we have specified the persistence-unit name for a PMF 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), e.g by query or by PM.getObjectById(). **Only applies if the underlying datastore supports it** {**0**, A positive value (MILLISECONDS)}
datanucleus.datastoreWriteTimeout	The timeout to apply to all writes (millisecs), e.g by makePersistent, or by an update. **Only applies if the underlying datastore supports it** {**0**, A positive value (MILLISECONDS)}
datanucleus.singletonPMFForName	Whether to only allow a singleton PMF for a particular name (the name can be either the name of the PMF in `jdoconfig.xml`, or the name of the persistence-unit). If a subsequent request is made for a PMF with a name that already exists then a warning will be logged and the original PMF returned. {true, **false**}
datanucleus.allowListenerUpdateAfterInit	Whether you want to be able to add/remove listeners on the JDO PMF after it is marked as not configurable (when the first PM is created). The default matches the JDO spec, not allowing changes to the listeners in use. {true, **false**}
datanucleus.cdi.bean.manager	Specifies a CDI `BeanManager` object that will be used to allow injection of dependencies into `AttributeConverter` objects.
datanucleus.jmxType	Which JMX server to use when hooking into JMX. Please refer to the Monitoring Guide {default, mx4j}
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> elements, 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 *pm.getObjectById(id)* be translated into valid JDO ids if there is a suitable translator. See Identity String Translator Plugin
datanucleus.identityKeyTranslatorType	You can allow identities input to *pm.getObjectById(cls, key)* be translated into valid JDO ids if there is a suitable key translator. See Identity Key Translator Plugin
datanucleus.datastoreIdentityType	Which "datastore-identity" class plugin to use to represent datastore identities. Refer to Datastore Identity extensions for details. {**datanucleus**, kodo, xcalia, {user-supplied plugin}}

Parameter	Description + Values
datanucleus.executionContext.maxIdle	Specifies the maximum number of ExecutionContext objects that are pooled ready for use {**20**, integer value greater than 0}
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 {**false**, true}
datanucleus.executionContext.closeActiveTxAction	Defines the action if a PM is closed and there is an active transaction present {**rollback**, exception}
datanucleus.objectProvider.className	Class name for the ObjectProvider to use when managing object state. The default for RDBMS is ReferentialStateManagerImpl, and is StateManagerImpl for all other datastores.
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.useImplementationCreator	Whether to allow use of the implementation-creator (feature of JDO to dynamically create implementations of persistent interfaces). {**true**, false}
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. You can set it to *false* if you always set both sides of a relation when persisting/updating. {**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 a JDOUserException is thrown. You can set it to *false* if you want to omit all consistency checks. {**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. For performance improvements, consider turning this off. {**true**, false}
datanucleus.classLoaderResolverName	Name of a ClassLoaderResolver to use in class loading. DataNucleus provides a default that loosely follows the JDO specification for class loading. This property allows the user to override this 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 JDO specification.

Parameter	Description + Values
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. {**false**, true}
datanucleus.findObject.validateWhenCached	When a user calls getObjectById (JDO) and they request validation this allows the turning off of validation when an object is found in the (L2) cache. Can be useful for performance reasons, but should be used with care. {**true**, false}
datanucleus.findObject.typeConversion	When calling PM.getObjectById(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). {**true**, false}

DataNucleus Schema Properties

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

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**}

Parameter	Description + Values
datanucleus.autoCreate WarnOnError	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.val idateAll	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.val idateTables	Whether to validate tables against the persistence definition. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.schema.val idateColumns	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**}
datanucleus.schema.val idateConstraints	Whether to validate table constraints against the persistence definition. Please refer to the Schema Guide for more details. {true, **false**}
datanucleus.readOnlyD atastore	Whether the datastore is read-only or not (fixed in structure and contents). {true, **false**}
datanucleus.readOnlyD atastoreAction	What happens when a datastore is read-only and an object is attempted to be persisted. {**EXCEPTION**, IGNORE}
datanucleus.generateSc hema.database.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.generateSc hema.scripts.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.generateSc hema.scripts.create.tar get	Name of the script file to write to if doing a "create" with the target as "scripts" {**datanucleus-schema-create.ddl**, {filename}}
datanucleus.generateSc hema.scripts.drop.targe t	Name of the script file to write to if doing a "drop" with the target as "scripts" {**datanucleus-schema-drop.ddl**, {filename}}
datanucleus.generateSc hema.scripts.create.sou rce	Name of a script file to run to create tables. Can be absolute filename, or URL string
datanucleus.generateSc hema.scripts.drop.sour ce	Name of a script file to run to drop tables. Can be absolute filename, or URL string
datanucleus.generateSc hema.scripts.load	Name of a script file to run to load data into the schema. Can be absolute filename, or URL string

Parameter	Description + Values
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, {user-plugin-name}}
datanucleus.identifier.namingFactory	Name of the identifier NamingFactory to use when generating table/column names etc (non-RDBMS datastores). {**datanucleus2**, jpa, {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)
datanucleus.identifier.tableSuffix	Suffix to be appended to all generated table names (if the identifier factory supports it)
datanucleus.defaultInheritanceStrategy	How to choose the inheritance strategy default for classes where no strategy has been specified. With *JDO2* this will be "new-table" for base classes and "superclass-table" for subclasses. With *TABLE_PER_CLASS* this will be "new-table" for all classes. {**JDO2**, TABLE_PER_CLASS}
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 PersistenceManagerFactory.

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 PM factories. 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}

Parameter	Description + Values
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 extension. 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**}
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 PM. 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 PMs 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. *MANUAL* means that operations will be sent only on flush()/commit(). *QUERY* means that operations will be sent on flush()/commit() and just before query execution. *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.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, Proxool, 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, Proxool, BoneCP, HikariCP, Tomcat, {others}}

Parameter	Description + Values
datanucleus.connection PoolingType.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, Proxool, BoneCP, HikariCP, Tomcat, {others}}
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 PM). {**true**, false}
datanucleus.connection .singleConnectionPerEx ecutionContext	With a PM 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 PM. {true, **false**}
datanucleus.connection .resourceType	Resource Type for primary connection {JTA, RESOURCE_LOCAL}
datanucleus.connection .resourceType2	Resource Type for secondary connection {JTA, RESOURCE_LOCAL}

DataNucleus Cache Properties

Extension

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

Parameter	Description + Values
datanucleus.cache.colle ctions	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.colle ctions.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}
datanucleus.cache.level 1.type	Name of the type of Level 1 cache to use. Defines the backing map. See also the Level 1 Cache docs {**soft**, weak, strong, {your-plugin-name}}

Parameter	Description + Values
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 the Level 2 Cache docs {none, **soft**, weak, javax.cache, coherence, ehcache, ehcacheclassbased, cacheonix, oscache, redis, 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 the Level 2 Cache docs {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}
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 Level 2 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 PMF 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 PMFs 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 expiry 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}
datanucleus.cache.level2.statisticsEnabled	With javax.cache L2 caches you can configure the cache to enable statistics gathering (accessible via JMX) {**false**, true}

Parameter	Description + Values
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 PMF 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

Extension

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

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
datanucleus.validation.group.pre-remove	The classes to validation on pre-remove callback

Parameter	Description + Values
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 PersistenceManagerFactory.

Parameter	Description + Values
datanucleus.valuegeneration.transactionAttribute	Whether to use the PM 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 and the transaction guide {read-uncommitted, **read-committed**, repeatable-read, serializable}
datanucleus.valuegeneration.sequence.allocationSize	Sets the default allocation size for any "sequence" value strategy. You can configure each member strategy individually but they fall back to this value if not set. {10, (integer value)}
datanucleus.valuegeneration.increment.allocationSize	Sets the default allocation size for any "increment" value strategy. You can configure each member strategy individually but they fall back to this value if not set. {10, (integer value)}

DataNucleus Metadata Properties

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

Parameter	Description + Values
datanucleus.metadata.jdoFileExtension	Suffix for JDO MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF's. {**jdo**, {file suffix}}
datanucleus.metadata.ormFileExtension	Suffix for ORM MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF's. {**orm**, {file suffix}}

Parameter	Description + Values
datanucleus.metadata.jdoqueryFileExtension	Suffix for JDO Query MetaData files. Provides the ability to override the default suffix and also to have one PMF with one suffix and another with a different suffix, hence allowing differing persistence of the same classes using different PMF's. {**jdoquery**, {file suffix}}
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**}
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.xml.allowJDO1_0	Whether we should allow XML metadata to be specified in locations from the JDO 1.0.0 spec. {**false**, true}
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.autoregistration	Whether to use the JDO auto-registration of metadata. Turned on by default {**true**, false}
datanucleus.metadata.supportORM	Whether to support "orm" mapping files. By default we use what the datastore plugin supports. This can be used to turn it off when the datastore supports it but we dont plan on using it (for performance) {**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. {**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*

Parameter	Description + Values
datanucleus.metadata.useDiscriminatorForSingleTable	With JPA 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 JDO `@Column` equivalent. {**false**, true}

DataNucleus Autostart Properties

 Extension

DataNucleus provides the following properties for configuring auto-start mechanism handling used by the PersistenceManagerFactory.

Parameter	Description + Values
datanucleus.autoStartMechanism	How to initialise DataNucleus at startup. This allows DataNucleus to read in from some source the classes that it was persisting for this data store the previous time. *XML* stores the information in an XML file for this purpose. *SchemaTable* (only for RDBMS) stores a table in the RDBMS for this purpose. *Classes* looks at the property *datanucleus.autoStartClassNames* for a list of classes. *MetaData* looks at the property *datanucleus.autoStartMetaDataFiles* for a list of metadata files The other option (default) is *None* (start from scratch each time). Please refer to the Auto-Start Mechanism Guide for more details. **Alternatively just use `persistence.xml` to specify the classes and/or mapping files to load at startup.** Note also that "Auto-Start" is for RUNTIME use only (not during SchemaTool). {**None**, XML, Classes, MetaData, SchemaTable}
datanucleus.autoStartMechanismMode	The mode of operation of the auto start mode. Currently there are 3 values. "Quiet" means that at startup if any errors are encountered, they are fixed quietly. "Ignored" means that at startup if any errors are encountered they are just ignored. "Checked" means that at startup if any errors are encountered they are thrown as exceptions. {Checked, Ignored, **Quiet**}
datanucleus.autoStartMechanismXmlFile	Filename used for the XML file for AutoStart when using "XML" Auto-Start Mechanism
datanucleus.autoStartClassNames	This property specifies a list of classes (comma-separated) that are loaded at startup when using the "Classes" Auto-Start Mechanism.
datanucleus.autoStartMetaDataFiles	This property specifies a list of metadata files (comma-separated) that are loaded at startup when using the "MetaData" Auto-Start Mechanism.

DataNucleus Query Properties

 Extension

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

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.closeable	When set to false (the default) will simply close all results when close() is called. When set to true it will also close the query object making it unusable, releasing all resources as well. Also applies to a JDO Extent use of close(). {true, **false**}
datanucleus.query.useFetchPlan	Whether to use the FetchPlan when executing a JDOQL query. The default is to use it which means that the relevant fields of the object will be retrieved. This allows the option of just retrieving the identity columns. {**true**, false}
datanucleus.query.compileOptimiseVarThis	This optimisation will detect and try to fix a query clause like "var == this" (which is pointless). It is not very advanced but may help in some situations {true, **false**}
datanucleus.query.jdoql.allowAll	javax.jdo.query.JDOQL queries are allowed by JDO only to run SELECT queries. This extension permits to bypass this limitation so that DataNucleus extension bulk "update" and bulk "delete" can be run. {**false**, true}
datanucleus.query.sql.allowAll	javax.jdo.query.SQL queries are allowed by JDO only to run SELECT queries. This extension permits to bypass this limitation (so for example can execute stored procedures). {**false**, true}
datanucleus.query.jpql.allowRange	JPQL queries, by the JPA 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}
datanucleus.query.checkUnusedParameters	Whether to check for unused input parameters and throw an exception if found. The JDO spec requires this check and is a good guide to having misnamed a parameter name in the query for example. {**true**, false}
datanucleus.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 PersistenceManagerFactory.

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.
datanucleus.rdbms.useLegacyNativeValueStrategy	This property changes the process for deciding the value strategy to use when the user has selected "native" to be like it was with DN version 3.0 and earlier, so using "increment" and "uuid-hex". {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 {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. {**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. {**true**, false}
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. {**ALL**, PK, NONE}
datanucleus.rdbms.classAdditionMaxRetries	The maximum number of retries when trying to find a class to persist or when validating a class. {**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). {**DataNucleus**, JDO2}
datanucleus.rdbms.uniqueConstraints.mapInverse	Whether to add unique constraints to the element table for a map inverse field. {**true**, false}
datanucleus.rdbms.discriminatorPerSubclassTable	Property that controls if only the base class where the discriminator is defined will have a discriminator column {**false**, true}
datanucleus.rdbms.stringDefaultLength	The default (max) length to use for all strings that don't have their column length defined in MetaData. {**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. {**EXCEPTION**, TRUNCATE}

Parameter	Description + Values
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*. {**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). {true, **false**}
datanucleus.rdbms.query.fetchDirection	The direction in which the query results will be navigated. {**forward**, reverse, unknown}
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. {**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. {**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. {**exists**, none}
datanucleus.rdbms.oracle.nlsSortOrder	Sort order for **Oracle** String fields in queries (BINARY disables native language sorting). {**LATIN**, See Oracle documentation}
datanucleus.rdbms.mysql.engineType	Specify the default engine for any tables created in **MySQL/MariaDB**. {**InnoDB**, valid engine for MySQL}
datanucleus.rdbms.mysql.collation	Specify the default collation for any tables created in **MySQL/MariaDB**. {valid collation for MySQL}
datanucleus.rdbms.mysql.characterSet	Specify the default charset for any tables created in **MySQL/MariaDB**. {valid charset for MySQL}
datanucleus.rdbms.informix.useSerialForIdentity	Whether we are using SERIAL for identity columns with **Informix** (instead of SERIAL8). {true, **false**}
datanucleus.rdbms.schemaTable.tableName	Name of the table to use when using auto-start mechanism of "SchemaTable" Please refer to the [Auto-Start guide](#) {NUCLEUS_TABLES, Valid table name}
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. {true, **false**}

Parameter	Description + Values
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 amount of time. Setting this to true means that your datastore schema has to be correct and no checks will be performed. {true, **false**}
datanucleus.rdbms.refreshAllTablesOnRefreshColumns	Whether to refresh all known tables whenever we need to get schema info for a table from the JDBC driver. Set this to *true* if you want to refresh all known table's information in the same call. If your application is changing the schema often then this should likely be *false*. {true, **false**}
datanucleus.rdbms.sqlTableNamingStrategy	Name of the plugin to use for defining the names of the aliases of tables in SQL statements. {**alpha-scheme**, t-scheme}
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 {**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. {true, **false**}
datanucleus.rdbms.statementLogging	How to log SQL statements. The default is to log the statement and replace any parameters with the value provided in angle brackets. Alternatively you can log the statement with any parameters replaced by just the values (no brackets). The final option is to log the raw JDBC statement (with ? for parameters). {**values-in-brackets**, values, jdbc}
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. {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. {**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. {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. {true, **false**}

Parameter	Description + Values
datanucleus.cassandra. compression	Type of compression to use for the Cassandra cluster. {**none**, snappy}
datanucleus.cassandra. metrics	Whether metrics are enabled for the Cassandra cluster. {**true**, false}
datanucleus.cassandra. ssl	Whether SSL is enabled for the Cassandra cluster. {true, **false**}
datanucleus.cassandra. socket.readTimeoutMill is	Socket read timeout for the Cassandra cluster.
datanucleus.cassandra. socket.connectTimeout Millis	Socket connect timeout for the Cassandra cluster.

Closing PersistenceManagerFactory

Since the PMF 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

```
pmf.close();
```

Level 2 Cache

The *PersistenceManagerFactory* has an optional cache of all objects across all _PersistenceManager_s. This cache is called the **Level 2 (L2) cache**, and JDO 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 JDO 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 JDO 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 - an 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 **javax.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 PersistenceManager. 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.

Plugin The L2 cache is a DataNucleus plugin point 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 JDO-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 JDO and DataNucleus you have the ability to control which objects remain in the cache. This is available via a method on the *PersistenceManagerFactory*.

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
DataStoreCache cache = pmf.getDataStoreCache();
```

The *DataStoreCache* interface *Javadoc* provides methods to control the retention of objects in the cache. You have 3 groups of methods

- **evict** - used to remove objects from the L2 Cache

- **pin** - used to pin objects into the cache, meaning that they will not get removed by garbage collection, and will remain in the L2 cache until removed.

- **unpin** - used to reverse the effects of pinning an object in the L2 cache. This will mean that the object can thereafter be garbage collected if not being used.

These methods can be called to *pin* objects into the cache that will be much used. Clearly this will be very much application dependent, but it provides a mechanism for users to exploit the caching features of JDO. If an object is not "pinned" into the L2 cache then it can typically be garbage collected at any time, so you should utilise the pinning capability for objects that you wish to retain access to during your application lifetime. For example, if you have an object that you want to be found from the cache you can do

```java
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(props);
DataStoreCache cache = pmf.getDataStoreCache();
cache.pinAll(MyClass.class, false); // Pin all objects of type MyClass from now on
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    pm.makePersistent(myObject);
    // "myObject" will now be pinned since we are pinning all objects of type MyClass.

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

Thereafter, whenever something refers to *myObject*, it will find it in the L2 cache. To turn this behaviour off, the user can either **unpin** it or **evict** it.

JDO allows control over which classes are put into a L2 cache. You do this by specifying the **cacheable** attribute to *false* (defaults to true). So with the following specification, no objects of type *MyClass* will be put in the L2 cache.

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

or using XML metadata

```
<class name="MyClass" cacheable="false">
    ...
</class>
```

JDO allows you control over which fields of an object are put in the L2 cache. You do this by specifying the **cacheable** attribute to *false* (defaults to true). This setting is only required for fields that are relationships to other persistable objects. Like this

```
public class MyClass
{
    ...

    Collection values;

    @Cacheable("false")
    Collection elements;
}
```

or using XML metadata

```
<class name="MyClass">
    <field name="values"/>
    <field name="elements" cacheable="false"/>
    ...
</class>
```

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 L2 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 L2 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 L2 cache for this field of this object

When pulling an object in from the L2 cache and it has a reference to another object DataNucleus uses the "identity" to find that object in the L1 or L2 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.expireMillis=...
datanucleus.cache.level2.memcached.servers=...
```

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={expiry-in-millis}
datanucleus.cache.level2.configurationFile={Cacheonix configuration file (in
classpath)}
```

and define a *cacheonix-config.xml* like

```xml
<?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.expiryMillis={expiry-in-millis}
datanucleus.cache.level2.redis.database={database, or use the default '1'}
datanucleus.cache.level2.redis.timeout={optional 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

```
JDODataStoreCache cache = (JDODataStoreCache)pmf.getDataStoreCache();
NamedCache tangosolCache = ((TangosolLevel2Cache)cache.getLevel2Cache
()).getTangosolCache();
```

Level 2 Cache implementation

Objects in a Level 2 cache are keyed by their JDO "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.
- 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 JDO allows you to generate the schema for your *persistence-unit* when creating a PMF. 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).

- **datanucleus.generateSchema.database.mode** which can be set to *create, drop, drop-and-create* or *none* to control the generation of the schema in the database.

- **datanucleus.generateSchema.scripts.mode** which can be set to *create, drop, drop-and-create* or *none* to control the generation of the schema as scripts (DDL). See also *datanucleus.generateSchema.scripts.create.target* and *datanucleus.generateSchema.scripts.drop.target* which will be generated using this mode of operation.

- **datanucleus.generateSchema.scripts.create.target** - this should be set to the name of a DDL script file that will be generated when using *datanucleus.generateSchema.scripts.mode*

- **datanucleus.generateSchema.scripts.drop.target** - this should be set to the name of a DDL script file that will be generated when using *datanucleus.generateSchema.scripts.mode*

- **datanucleus.generateSchema.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)

- **datanucleus.generateSchema.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)

- **datanucleus.generateSchema.scripts.load** - set this to an SQL script of your own that will insert any data that you require to be available when your PMF is initialised

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).

> **ℹ** 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).

> **ℹ** 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.

> **ℹ** 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. There is JDO metadata attribute that

allows you to specify the order of columns for schema generation; it is achieved by specifying the metadata attribute *position* against the column.

```
<column position="1"/>
```

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

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 PMF as "read-only". You do this by setting the persistence property **javax.jdo.option.ReadOnly** to *true*.

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

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 **datanucleus.generateSchema** 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).
metadataDirectory	${project.build.output Directory}	Directory to use for schema generation files (classes/mappings)
metadataIncludes	**/**.jdo, **/**.class	Fileset to include for schema generation

Property	Default	Description
metadataExcludes		Fileset to exclude for schema generation
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)
props		Name of a properties file for the datastore (PMF)
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)
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)
includeAutoStart	false	Whether to include auto-start mechanisms in SchemaTool usage

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>5.0.2</version>
            <configuration>
                <props>${basedir}/datanucleus.properties</props>
                <log4jConfiguration>${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 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.	**JDO**, JPA
props	The filename to use for persistence properties	
persistenceUnit	Name of the persistence-unit that we should manage the schema for (defines the classes and the properties defining the datastore).	

Parameter	Description	values
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**
includeAutoStart	Whether to include any auto-start mechanism in SchemaTool usage	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	Mandatory
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	✔
datanucleus.Mapping	ORM Mapping name	✖
log4j.configuration	Log4J 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.

You define the JDO files to create the tables using **fileset**.

```
<taskdef name="schematool" classname="org.datanucleus.store.schema.SchemaToolTask" />

<schematool failonerror="true" verbose="true" mode="create">
    <classpath>
        <path refid="schematool.classpath"/>
    </classpath>
    <fileset dir="${classes.dir}">
        <include name="**/*.jdo"/>
    </fileset>
    <sysproperty key="datanucleus.ConnectionURL" value=
"${datanucleus.ConnectionURL}"/>
    <sysproperty key="datanucleus.ConnectionUserName"
value="${datanucleus.ConnectionUserName}"/>
    <sysproperty key="datanucleus.ConnectionPassword"
value="${datanucleus.ConnectionPassword}"/>
    <sysproperty key="datanucleus.Mapping" value="${datanucleus.Mapping}"/>
</schematool>
```

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)
    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)
        -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
        -includeAutoStart : whether to include any auto-start mechanism in SchemaTool
usage
        -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-jdo.jar
lib/datanucleus-rdbms.jar, lib/datanucleus-hbase.jar,  etc
lib/javax.jdo.jar
lib/mysql-connector-java.jar (driver for the datastore, whether RDBMS, HBase etc)
log4j.properties
```

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/javax.jdo.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
      -props datanucleus.properties
      target/classes/org/datanucleus/examples/normal/package.jdo
      target/classes/org/datanucleus/examples/inverse/package.jdo


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-jdo.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/datanucleus-rdbms.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/javax.jdo.jar
>>  /home/andy/work/DataNucleus/samples/packofcards/lib/mysql-connector-java.jar

DataNucleus SchemaTool : Input Files
>>
/home/andy/work/DataNucleus/samples/packofcards/target/classes/org/datanucleus/example
s/inverse/package.jdo
>>
/home/andy/work/DataNucleus/samples/packofcards/target/classes/org/datanucleus/example
s/normal/package.jdo

DataNucleus SchemaTool : Taking JDO properties from file "datanucleus.properties"

SchemaTool completed successfully
```

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
{
    public int createDatabase(String catalogName, String schemaName, Properties
props);
    public int deleteDatabase(String catalogName, String schemaName, Properties
props);

    public int createSchemaForClasses(Set<String> classNames, Properties props);
    public int deleteSchemaForClasses(Set<String> classNames, Properties props);
    public int 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

```
JDOPersistenceManagerFactory pmf =
    (JDOPersistenceManagerFactory)JDOHelper.getPersistenceManagerFactory
("datanucleus.properties");
PersistenceNucleusContext ctx = pmf.getNucleusContext();
...
List classNames = new ArrayList();
classNames.add("mydomain.A");
classNames.add("mydomain.B");
try
{
    Properties props = new Properties();
    // Set any properties for schema generation
    ((SchemaAwareStoreManager)ctx.getStoreManager()).createSchemaForClasses(
classNames, props);
}
catch(Exception e)
{
    ...
}
```

Schema Adaption

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

```java
import org.datanucleus.api.jdo.JDOPersistenceManagerFactory;
import org.datanucleus.store.StoreManager;
import org.datanucleus.store.schema.StoreSchemaHandler;

[assumed to have "pmf"]
...

StoreManager storeMgr = ((JDOPersistenceManagerFactory)pmf).getStoreManager();
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

RDBMS : Datastore Types Information

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

```java
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* 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();

    ...
}
```

AutoStart Mechanism

Extension

By default with JDO implementations when you open a *PersistenceManagerFactory* and obtain a *PersistenceManager* DataNucleus knows nothing about which classes are to be persisted to that datastore (unless you created the PMF using a persistence-unit). JDO implementations only load the Meta-Data for any class when the class is first enlisted in a *PersistenceManager* operation. For example you call *makePersistent* on an object. The first time a particular class is encountered DataNucleus will dynamically load the Meta-Data for that class. This typically works well since in an application in a particular operation the *PersistenceManagerFactory* may well not encounter all classes that are persistable to that datastore. The reason for this dynamic loading is that JDO implementations can't be expected to scan through the whole Java CLASSPATH for classes that could be persisted there. That would be inefficient.

There are situations however where it is desirable for DataNucleus to have knowledge about what is to be persisted, or what subclasses of a candidate are possible on executing a query, so that it can load the Meta-Data at initialisation of the persistence factory and hence when the classes are encountered for the first time nothing needs doing. There are several ways of achieving this

- Define your classes/MetaData in a Persistence Unit and when the *PersistenceManagerFactory* is initialised it loads the persistence unit, and hence the MetaData for the defined classes and mapping files. **This is standardised, and hence is the recommended way.**

- Put a `package.jdo` at the root of the CLASSPATH, containing all classes, and when the first class is encountered it searches for its metadata, encounters and parses the root `package.jdo`, and consequently loads the metadata for all classes mentioned in that file.

- Use a DataNucleus extension known as **Auto-Start Mechanism**. This is set with the persistence property **datanucleus.autoStartMechanism**. This can be set to *None*, *XML*, *Classes*, *MetaData*. In addition we have *SchemaTable* for RDBMS datastores. These are described below.

AutoStartMechanism : None

With this property set to "None" DataNucleus will have no knowledge about classes that are to be persisted into that datastore and so will add the classes when the user utilises them in calls to the various *PersistenceManager* methods.

AutoStartMechanism : XML

With *XML*, DataNucleus stores the information for starting up DataNucleus in an XML file. This is, by default, located in `datanucleusAutoStart.xml` in the current working directory. The file name can be configured using the persistence property **datanucleus.autoStartMechanismXmlFile**. The file is read at startup and DataNucleus loads the classes using this information.

If the user changes their persistence definition a problem can occur when starting up DataNucleus. DataNucleus loads up its existing data from the XML configuration file and finds that a table/class required by the this file data no longer exists. There are 3 options for what DataNucleus will do in this situation. The property **datanucleus.autoStartMechanismMode** defines the behaviour of

DataNucleus for this situation.

- **Checked** will mean that DataNucleus will throw an exception and the user will be expected to manually fix their database mismatch (perhaps by removing the existing tables).

- **Quiet** (the default) will simply remove the entry from the XML file and continue without exception.

- **Ignored** will simply continue without doing anything.

See the DTD at this link. A sample file would look something like

```
<datanucleus_autostart>
    <class name="mydomain.MyClass" table="MY_TABLE_1" type="FCO" version="3.1.1"/>
</datanucleus_autostart>
```

AutoStartMechanism : Classes

With *Classes*, the user provides to the persistence factory the list of classes to use as the initial list of classes to be persisted. They specify this via the persistence property **datanucleus.autoStartClassNames**, specifying the list of classes as comma-separated. This gives DataNucleus a head start meaning that it will not need to "discover" these classes later.

AutoStartMechanism : MetaData

With *MetaData*, the user provides to the persistence factory the list of metadata files to use as the initial list of classes to be persisted. They specify this via the persistence property *datanucleus.autoStartMetaDataFiles*, specifying the list of metadata files as comma-separated. This gives DataNucleus a head start meaning that it will not need to "discover" these classes later.

AutoStartMechanism : SchemaTable (RDBMS only)

When using an RDBMS datastore the *SchemaTable* auto-start mechanism stores the list of classes (and their tables, types and version of DataNucleus) in a datastore table **NUCLEUS_TABLES**. This table is read at startup of DataNucleus, and provides DataNucleus with the necessary knowledge it needs to continue persisting these classes. This table is continuously updated during a session of a DataNucleus-enabled application.

If the user changes their persistence definition a problem can occur when starting up DataNucleus. DataNucleus loads up its existing data from NUCLEUS_TABLES and finds that a table/class required by the NUCLEUS_TABLES data no longer exists. There are 3 options for what DataNucleus will do in this situation. The property **datanucleus.autoStartMechanismMode** defines the behaviour of DataNucleus for this situation.

- **Checked** will mean that DataNucleus will throw an exception and the user will be expected to manually fix their database mismatch (perhaps by removing the existing tables).

- **Quiet** (the default) will simply remove the entry from NUCLEUS_TABLES and continue without exception.

- **Ignored** will simply continue without doing anything.

The default database schema used the *SchemaTable* is described below:

```
TABLE : NUCLEUS_TABLES
(
    COLUMN : CLASS_NAME VARCHAR(128) PRIMARY KEY,  -- Fully qualified persistent Class
name
    COLUMN : TABLE_NAME VARCHAR(128),              -- Table name
    COLUMN : TYPE VARCHAR(4),                      -- FCO | SCO
    COLUMN : OWNER VARCHAR(2),                     -- 1 | 0
    COLUMN : VERSION VARCHAR(20),                  -- DataNucleus version
    COLUMN : INTERFACE_NAME VARCHAR(255)           -- Fully qualified persistent Class
type
                                                   -- of the persistent Interface

implemented
)
```

If you want to change the table name (from NUCLEUS_TABLES) you can set the persistence property
datanucleus.rdbms.schemaTable.tableName

PersistenceManager

Now that we have our [PersistenceManagerFactory](#), providing the connection for our persistence context to our datastore, we need to obtain a *PersistenceManager* (PM) to manage the persistence of objects. Here we describe the majority of operations that you will are likely to need to know about.

> A *PersistenceManagerFactory* is designed to be thread-safe. A *PersistenceManager* is not. Note that if you set the persistence property **javax.jdo.option.Multithreaded** this acts as a hint to the PMF to provide *PersistenceManager(s)* that are usable with multiple threads. While DataNucleus makes efforts to make this *PersistenceManager* usable with multiple threads, it is not guaranteed to work multi-threaded in all situations, particularly around second class collection/map fields.

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

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

```
@PersistenceCapable
public class Person
{
    @PrimaryKey
    long id;

    String firstName;
    String lastName;
}

@PersistenceCapable
public class Account
{
    @PrimaryKey
    long id;

    Person person;
}
```

Opening/Closing a PersistenceManager

You obtain a *PersistenceManager* `Javadoc` as follows

```
PersistenceManager pm = pmf.getPersistenceManager();
```

You then perform all operations that you need using this *PersistenceManager* and finally you must **close** it. Forgetting to close it will lead to memory/resource leaks.

```
pm.close();
```

You likely will be performing the majority of operations on a *PersistenceManager* within a transaction, whether your transactions are controlled by a JavaEE container, by a framework such as Spring, or by locally defined transactions. Alternatively you can perform your operations non-transactional. 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 JDO-enabled application is persist your objects into the datastore. We have obtained a *PersistenceManager* to manage such interaction with the datastore, and now we persist our object

```
Person lincoln = new Person(1, "Abraham", "Lincoln");
pm.makePersistent(person);
```

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 *PersistentClean* (after makePersistent()), to *Hollow/Detached* (at commit).

Persisting multiple Objects in one call

When you want to persist multiple objects you simply call a different method on the *PersistenceManager*, like this

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

pm.makePersistentAll(coll);
```

As above, the objects are persisted to the datastore. The LifecycleState of the objects change from *Transient* to *PersistentClean* (after persist()), to *Hollow* (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. You can obtain the identity by calling

```
Object lincolnID = pm.getObjectId(lincoln);
```

Alternatively you can create an identity to represent this object by calling

```
Object lincolnID = pm.newObjectIdInstance(Person.class, 1);
```

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 lincoln = (Person)pm.getObjectById(lincolnID);
```

Extension

A DataNucleus extension is to pass in a String form of the identity to the above method. It accepts identity strings of the form

- *{fully-qualified-class-name}:{key}*
- *{discriminator-name}:{key}*

where the *key* is the identity value (datastore-identity) or the result of PK.toString() (application-identity). So for example we could input

```
Object obj = pm.getObjectById("mydomain.Person:1");
```

There is, of course, a bulk load variant too

```
Object[] objs = pm.getObjectsById(ids);
```

Extension

When you call the method *getObjectById* if an object with that identity is found in the cache then a call is, by default, made to validate it still exists. You can avoid this call to the datastore by setting the persistence property **datanucleus.findObject.validateWhenCached** to *false*.

Finding an object by its class and primary-key value

An alternate form of the *getObjectById* method is taking in the class of the object, and the "identity". This is for use where you have a *single field* that is primary key. Like this

```
Person lincoln = pm.getObjectById(Person.class, 1);
```

where 1 is the value of the primary key field (numeric).

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

> **ℹ** If the second argument is not of the type expected for the `@PrimaryKey` 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)

🔵 Extension

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 JDO API, however DataNucleus makes it available. Let's take an example

```
@PersistenceCapable
@Unique(name="MY_NAME_IDX", members={"firstName", "lastName"})
public class Person
{
    @PrimaryKey
    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

```
JDOPersistenceManager jdopm = (JDOPersistenceManager)pm;
Person p = jdopm.getObjectByUnique(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 = pm.getObjectById(Person.class, 1);  // Retrieves the object to delete
pm.deletePersistent(lincoln);
```

Don't forget that you can also use deletion by query to delete objects. Alternatively use bulk deletion.

Please note that when deleting a persist object the default is to **not** delete related objects.

Dependent Fields

If you want the deletion of a persistent object to cause the deletion of related objects then you need to mark the related fields in the mapping to be "dependent". For example with our example, if we modify it to be like this

```
@PersistenceCapable
public class Account
{
    ...

    @Persistent(dependent="true")
    Person person;
}
```

so now if we call

```
Account lincolnAcct = pm.getObjectById(Account.class, 1);  // Retrieves the Account to
delete
pm.deletePersistent(lincolnAcct);
```

This will delete the *Account* object as well as the *Person* account. The same applies on 1-N/M-N relations where you set the @Element, @Key, @Value dependent attribute accordingly. Some things to note about dependent fields.

- An object is deleted (using *deletePersistent()*) and that object has relations to other objects. If the other objects (either 1-1, 1-N, or M-N) are dependent then they are also deleted.

- An object has a 1-1 relation with another object, but the other object relation is nulled out. If the other object is dependent then it is deleted when the relation is nulled.

- An object has a 1-N collection relation with other objects and the element is removed from the collection. If the element is dependent then it will be deleted when removed from the collection. The same happens when the collections is cleared.

- An object has a 1-N map relation with other objects and the key is removed from the map. If the key or value are dependent and they are not present in the map more than once they will be deleted when they are removed. The same happens when the map is cleared.

Deletion using RDBMS Foreign Keys

With JDO you can use "dependent-field" as shown above. As an alternative (but not as a complement), when using RDBMS, you can use the datastore-defined foreign keys and let the datastore built-in "referential integrity" look after such deletions. DataNucleus provides a persistence property **datanucleus.deletionPolicy** allowing enabling of this mode of operation. The default setting of this property is "JDO2" which performs deletion of related objects as follows

- If *dependent-field* is true then use that to define the related objects to be deleted.

- Else, if the column of the foreign-key field is NULLable then NULL it and leave the related object alone

- Else deleted the related object (and throw exceptions if this fails for whatever datastore-related reason)

The other setting of this property is "DataNucleus" which performs deletion of related objects as follows

- If *dependent-field* is true then use that to define the related objects to be deleted

- If a *foreign-key* is specified (in MetaData) for the relation field then leave any deletion to the datastore to perform (or throw exceptions as necessary)

- Else, if the column of the foreign-key field is NULLable then NULL it and leave the related object alone

- Else deleted the related object (and throw exceptions if this fails for whatever datastore-related reason)

As you can see, with the second option you have the ability to utilise datastore "referential integrity" checking using your MetaData-specified `<foreign-key>` elements.

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 *PersistenceManager*) then when you are ready to persist the changes you do the following

```
Person updatedLincoln = pm.makePersistent(lincoln);
```

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

```
Person lincoln = pm.getObjectById(Person.class, 1); // "lincoln" is now managed by
"pm", and in "hollow/persistent-clean" 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 *PersistenceManager* method to push the changes. This is part of the mechanism known as *transparent persistence*.

> 💡 Don't forget that you can also use bulk update to update a group of objects of a type.

Detaching a persisted Object

As long as your persistable class is *detachable* (see the mapping guide) then you can *detach* objects of that type. Being *detached* means that your object is no longer managed by a particular *PersistenceManager* and hence usable in other tiers of your application. In this case you want to *detach* the object (and its related sub-objects) so that they can be passed across to the part of the application that requires it. To do this you do

```
Person detachedLincoln = pm.detachCopy(lincoln); // Returns a copy of the persisted
object, in detached state
```

The detached object is like the original object except that it has no StateManager connected, and it stores its JDO identity and version. It retains a list of all fields that are modified while it is detached. This means that when you want to "attach" it to the data-access layer it knows what to update.

Some things to be aware of with the *detachment* process.

- Calling *detachCopy* on an object that is not detachable will return a **transient** instance that is a COPY of the original, so use the COPY thereafter.

- Calling *detachCopy* on an object that is detachable will return a **detached** instance that is a COPY of the original, so use this COPY thereafter

- A *detached* object retains the id of its datastore entity. Detached objects should be used where you want to update the objects and attach them later (updating the associated object in the datastore. If you want to create copies of the objects in the datastore with their own identities you should use *makeTransient* instead of *detachCopy*.

- Calling *detachCopy* will detach all fields of that object that are in the current Fetch Groups for that class for that *PersistenceManager*.

- By default the fields of the object that will be detached are those in the *Default Fetch Group*.

- You should choose your Fetch Group carefully, bearing in mind which object(s) you want to access whilst detached. Detaching a relation field will detach the related object as well.

- If you don't detach a field of an object, you **cannot** access the value for that field while the object is detached.

- If you don't detach a field of an object, you **can** update the value for that field while detached, and thereafter you can access the value for that field.

Detaching objects used by a transaction

To make the detachment process transparent you can set the persistence property **datanucleus.DetachAllOnCommit** to true and when you commit your transaction all objects enlisted in the transaction will be detached. If you just want to apply this setting for a *PersistenceManager* then there is a *setDetachAllOnCommit* method on the *PersistenceManager*.

> ℹ️ This only has any effect when performing operations **in a transaction**.

Detach objects on close of the PersistenceManager

Extension

A further variation is known as "detachOnClose" and means that if enabled (setting persistence property **datanucleus.DetachOnClose** to *true*), when you close your *PersistenceManager* you are opting to have all instances currently cached in the Level 1 Cache of that *PersistenceManager* to be detached automatically.

> ⚠️ This will not work in a JavaEE environment when using JCA.

> ℹ️ It is recommended that you use "DetachAllOnCommit" instead of this wherever possible since that is standard JDO and would work in all JavaEE environments also.

Detached Fields

Extension

When an object is detached it is typically passed to a different layer of an application and potentially changed. During the course of the operation of the system it may be required to know what is loaded in the object and what is dirty (has been changed since detaching). DataNucleus provides an extension to allow interrogation of the detached object.

```
String[] loadedFieldNames = NucleusJDOHelper.getLoadedFields(obj, pm);
String[] dirtyFieldNames = NucleusJDOHelper.getDirtyFields(obj, pm);
```

So you have access to the names of the fields that were loaded when detaching the object, and also to the names of the fields that have been updated since detaching.

Serialization of Detachable classes

During enhancement of Detachable classes, a field called *jdoDetachedState* is added to the class definition. This field allows reading and changing tracking of detached objects while they are not managed by a PersistenceManager.

When serialization occurs on a Detachable object, the *jdoDetachedState* field is written to the serialized object stream. On deserialize, this field is written back to the new deserialized instance. This process occurs transparently to the application. However, if deserialization occurs with an un-enhanced version of the class, the detached state is lost.

Serialization and deserialization of Detachable classes and un-enhanced versions of the same class is only possible if the field *serialVersionUID* is added. It's recommended during development of the class, to define the *serialVersionUID* and make the class implement the *java.io.Serializable* interface.

Attaching a persisted Object

As you saw above, when we update an object in detached state we can update it in the datastore by *attaching* it to a *PersistenceManager*.

```
Person attachedLincoln = pm.makePersistent(lincoln); // Returns a copy of the detached
object, in attached state
```

Once the object is *attached* it is then managed by the *PersistenceManager*, and in *PersistentClean* state.

Some things to be aware of with the *attachment* process.

- Calling *makePersistent* will return an (attached) copy of the detached object. It will attach all fields that were originally detached, and will also attach any other fields that were modified whilst detached.

Copy On Attach

By default when you are attaching a detached object it will return an attached copy of the detached object. JDO provides a feature called *copy-on-attach* that allows this attachment to just migrate the existing detached object into attached state.

You enable this by setting the persistence property **datanucleus.CopyOnAttach** to *false*. Alternatively you can use the methods *PersistenceManagerFactory.setCopyOnAttach(boolean flag)* or *PersistenceManager.setCopyOnAttach(boolean flag)*. Consequently our attach code would become

```
pm.makePersistent(lincoln); // object "lincoln" is now in attached state after this
call
```

> if using this feature and you try to attach two detached objects representing the same underlying persistent object within the same transaction (i.e a persistent object with the same identity already exists in the level 1 cache), then a JDOUserException will be thrown.

Refresh of objects

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

```
pm.refresh(lincoln);
```

This will do the following

- Refresh the values of all FetchPlan fields in the object
- Unload all non-FetchPlan fields in the object

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

Cascading Operations

When you have relationships between entities, and you persist one entity, by default the related entity *will* be persisted. This is referred to as **persistence-by-reachability**.

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

```
pm.makePersistent(acct1);
```

This will persist the *Account* object and since it refers to the *Person* object, that will be persisted also.

 Extension

DataNucleus allows you to disable cascading of persist/update operations by using the `@Extension` metadata. So if we change our class like this

```
@PersistenceCapable
public class Account
{
    @PrimaryKey
    long id;

    @Extension(vendorName="datanucleus", key="cascade-persist", value="false")
    @Extension(vendorName="datanucleus", key="cascade-update", value="false")
    Person person;
}
```

now when we do this

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

it will not persist the related *Person* object (but will likely throw an exception due to it being present).

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

- Compound 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"
```

> ⚠️ With a *bidirectional* relation you must 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"
```

Reachability

With JDO, when you persist an object, all related objects (reachable from the fields of the object being persisted) will be persisted at the same time (unless already persistent). This is called *persistence-by-reachability*. For example

```
A a = new A();
B b = new B();
a.setB(b);
pm.makePersistent(a); // "a" and "b" are now provisionally persistent
```

This additionally applies when you have an object managed by the *PersistenceManager*, and you set a field to refer to a related object - this will make the related object provisionally persistent also. For example

```
A a = new A();
pm.makePersistent(a); // "a" is now provisionally persistent
B b = new B();
a.setB(b); // "b" is now provisionally persistent
```

Persistence-By-Reachability-At-Commit

An additional feature of JDO is the ability to re-run the *persistence-by-reachability* algorithm **at commit** so as to check whether the objects being made persistent should definitely be persisted. This is for the following situation.

- Start a transaction

- Persist object A. This persists related object B.

- Delete object A from persistence

Commit the transaction.

If you have persistence property **datanucleus.persistenceByReachabilityAtCommit** set to true (default) then this will recheck the persisted objects should remain persistent. In this case it will find B and realise that it was only persisted due to A (which has since been deleted), hence B will not remain persistent after the transaction. If you had property **datanucleus.persistenceByReachabilityAtCommit** set to false then B will remain persistent after the transaction.

> If you set this persistence property to *false* then this will give a speed benefit, since at commit it no longer has to re-check all reachability for subsequent deletions. Consequently, if you are sure you have not subsequently deleted an object you just persisted, you are advised to set this property to *false*.

Managed Relationships

As previously mentioned, users should really set both sides of a bidirectional relation. DataNucleus provides a good level of *managed relations* in that it will *attempt* to correct any missing information in relations to make both sides consistent. What it provides is defined below

For a *1-1 bidirectional relation*, at persist you should set one side of the relation and the other side will be set to make it consistent. If the respective sides are set to inconsistent objects then an exception will be thrown at persist. At update of owner/non-owner side the other side will also be updated to make them consistent.

For a *1-N bidirectional relation* and you only specify the element owner then the collection must be Set-based since DataNucleus cannot generate indexing information for you in that situation (you must position the elements). At update of element or owner the other side will also be updated to make them consistent. At delete of element the owner collection will also be updated to make them consistent. **If you are using a List you MUST set both sides of the relation**

For an *M-N bidirectional relation*, at persist you MUST set one side and the other side will be populated at commit/flush to make them consistent.

This management of relations can be turned on/off using a persistence property **datanucleus.manageRelationships**. If you always set both sides of a relation at persist/update then you could safely turn it off.

> 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.

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. You can alleviate this by changing how you flush/commit the persistent changes. You can do it like this, for

example

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();
    for (int i=0; i<100000; i++)
    {
        Wardrobe wardrobe = new Wardrobe();
        wardrobe.setModel("3 doors");
        pm.makePersistent(wardrobe);
        if (i % 10000 == 0)
        {
            // Flush every 10000 objects
            pm.flush();
        }
    }
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

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

Level 1 Cache

Each *PersistenceManager* 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.

- *none* - will turn off L1 caching. **Only ever use this where the cache is of no use and you are**

performing bulk operations and not requiring objects returned

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 objects from the L1 cache programmatically you should use the *pm.evict* or *pm.evictAll* methods.

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 *PersistenceManager*. When the *PersistenceManager* is closed the L1 cache is cleared.

Plugin The L1 cache is a DataNucleus plugin point allowing you to provide your own cache where you require it.

PersistenceManagerProxy

As we have already described for normal persistence, you perform all operations using a *PersistenceManager*, needing to obtain this when you want to start datastore operations.

In some architectures (e.g in a web environment) it can be convenient to maintain a single *PersistenceManager* for use in a servlet init() method to initialise a static variable. Alternatively for use in a SessionBean to initialise a static variable. The JDO API provides a "proxy" object that can be used for this purpose. Thereafter you just refer to the proxy. The proxy isn't the actual *PersistenceManager* just a proxy, delegating to the real object. If you call close() on the proxy the real PM will be closed, and when you next invoke an operation on the proxy it will create a new PM delegate and work with that.

To create a PM proxy is simple

```
PersistenceManager pm = pmf.getPersistenceManagerProxy();
```

So we have our proxy, and now we can perform operations in the same way as we do with any *PersistenceManager*.

Datastore Sequences API

Particularly when specifying the identity of an object, sequences are a very useful facility. DataNucleus supports the automatic assignment of sequence values for object identities. However such sequences may also have use when a user wishes to assign such identity values themselves, or for other roles within an application. JDO defines an interface for sequences for use in an application - known as **Sequence**. *Javadoc*. There are 2 forms of "sequence" available through this interface - the ones that DataNucleus provides utilising datastore capabilities, and ones that a user provides using something known as a "factory class".

DataNucleus Sequences

DataNucleus internally provides 2 forms of sequences. When the underlying datastore supports native sequences, then these can be leveraged through this interface. Alternatively, where the underlying datastore doesn't support native sequences, then a table-based incrementing sequence can be used. The first thing to do is to specify the **Sequence** in the Meta-Data for the package requiring the sequence. This is done as follows

```xml
<jdo>
    <package name="MyPackage">
        <class name="MyClass">
            ...
        </class>

        <sequence name="ProductSequence" datastore-sequence="PRODUCT_SEQ"
strategy="contiguous"/>
        <sequence name="ProductSequenceNontrans" datastore-
sequence="PRODUCT_SEQ_NONTRANS" strategy="nontransactional"/>
    </package>
</jdo>
```

So we have defined two **Sequences** for the package *MyPackage*. Each sequence has a symbolic name that is referred to within JDO (within DataNucleus), and it has a name in the datastore. The final attribute represents whether the sequence is transactional or not.

All we need to do now is to access the **Sequence** in our persistence code in our application. This is done as follows

```java
PersistenceManager pm = pmf.getPersistenceManager();

Sequence seq = pm.getSequence("MyPackage.ProductSequence");
```

and this **Sequence** can then be used to provide values.

```java
long value = seq.nextValue();
```

Please be aware that when you have a **Sequence** declared with a strategy of "contiguous" this means "transactional contiguous" and that you need to have a Transaction open when you access it.

JDO allows control over the allocation size (default=50) and initial value (default=1) for the sequence. So we can do

```xml
<sequence name="ProductSequence" datastore-sequence="PRODUCT_SEQ"
strategy="contiguous" allocation-size="10"/>
```

which will allocate 10 new sequence values each time the allocated sequence values is exhausted.

Factory Class Sequences

It is equally possible to provide your own **Sequence** capability using a *factory class*. This is a class that creates an implementation of the JDO **Sequence**. Let's give an example of what you need to provide. Firstly you need an implementation of the JDO **Sequence** interface, so we define ours like this

```java
public class SimpleSequence implements Sequence
{
    String name;
    long current = 0;

    public SimpleSequence(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public Object next()
    {
        current++;
        return new Long(current);
    }

    public long nextValue()
    {
        current++;
        return current;
    }

    public void allocate(int arg0)
    {
    }

    public Object current()
    {
        return new Long(current);
    }

    public long currentValue()
    {
        return current;
    }
}
```

So our sequence simply increments by 1 each call to *next()*. The next thing we need to do is provide

a *factory class* that creates this **Sequence**. This factory needs to have a static *newInstance* method that returns the **Sequence** object. We define our factory like this

```java
package org.datanucleus.samples.sequence;

import javax.jdo.datastore.Sequence;

public class SimpleSequenceFactory
{
    public static Sequence newInstance()
    {
        return new SimpleSequence("MySequence");
    }
}
```

and now we define our MetaData like this

```xml
<jdo>
    <package name="MyPackage">
        <class name="MyClass">
            ...
        </class>

        <sequence name="ProductSequenceFactory" strategy="nontransactional"
            factory-class="org.datanucleus.samples.sequence.SimpleSequenceFactory"/>
    </package>
</jdo>
```

So now we can call

```java
PersistenceManager pm = pmf.getPersistenceManager();

Sequence seq = pm.getSequence("MyPackage.ProductSequenceFactory");
```

Object Lifecycle

During the persistence process, an object goes through lifecycle changes. Below we demonstrate the primary object lifecycle changes for JDO. JDO has a very high degree of flexibility and so can be configured to operate in different modes. The mode most consistent with JPA is shown below (this has the persistence property **datanucleus.DetachAllOnCommit** set to *true*).
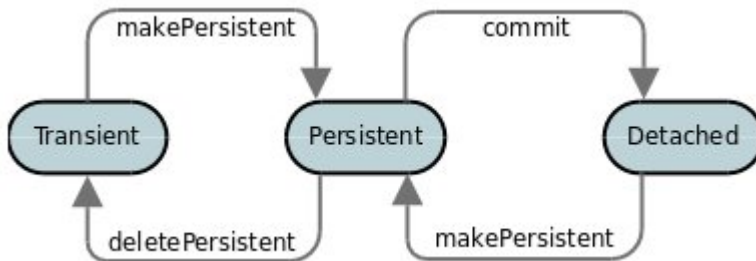
So a newly created object is **transient**. You then persist it and it becomes **persistent**. You then commit the transaction and it is detached for use elsewhere in the application. You then attach any changes back to persistence and it becomes **persistent** again. Finally when you delete the object from persistence and commit that transaction it is in **transient** state.

An alternative JDO lifecycle occurs when you have **datanucleus.DetachAllOnCommit** as *false*. Now at commit the object moves into **hollow** state (still has its identity, but its field values are optionally unloaded). Set the persistence property **datanucleus.RetainValues** to not unset the values of any non-primary-key fields when migrating to **hollow** state.
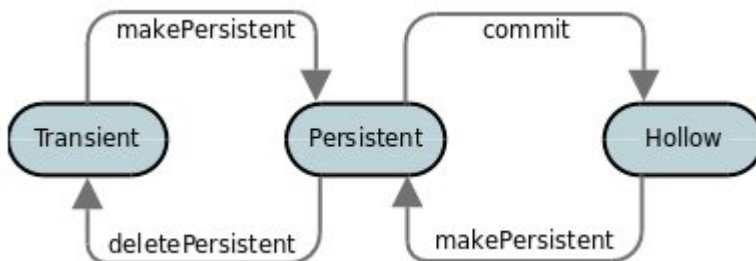
With JDO there are actually some additional lifecycle states, notably when an object has a field changed, becoming *dirty*, so you get an object in "persistent-dirty", "detached-dirty" states for example. The average user doesn't need to know about these so we don't cover them here.

See also :-

* Attach/Detach of objects

Helper Methods

To inspect the lifecycle state of an object, simply call

```
ObjectState state = JDOHelper.getObjectState(obj);
```

JDO provides a series of other helper methods for lifecycle operations. These are documented on

the Apache JDO site.

Further to this DataNucleus provides yet more helper methods

```
String[] fieldNames = NucleusJDOHelper.getDirtyFields(pc, pm);
String[] fieldNames = NucleusJDOHelper.getLoadedFields(pc, pm);
```

These methods returns the names of the dirty/loaded fields in the supplied object. The *pm* argument is only required if the object is detached

```
Boolean dirty = NucleusJDOHelper.isDirty(pc, "fieldName", pm);
Boolean loaded = NucleusJDOHelper.isLoaded(pc, "fieldName", pm);
```

These methods returns whether the specified field in the supplied object is dirty/loaded. The *pm* argument is only required if the object is detached

Transactions

Persistence operations performed by the *PersistenceManager* 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 JDO Transaction API

- JTA transactions : managed using the JTA UserTransaction API, or using the JDO Transaction API

- Container-managed transactions : managed by a JavaEE environment

- Spring-managed transactions : managed by SpringFramework

- No transactions : "auto-commit" mode

- Flushing a Transaction

- Controlling transaction isolation level

- Synchronising with transaction commit

- Read-Only transactions

- RDBMS : Savepoints

Locally-Managed Transactions

When using a JDO implementation such as DataNucleus in a JavaSE environment, the transactions are by default **Locally Managed Transactions**. The users code will manage the transactions by starting, committing or rolling back the transaction itself. With these transactions with JDO you would do something like

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    {users code to persist objects}

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

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

JTA Transactions

When using a JDO implementation such as DataNucleus in a JavaSE environment, you can also make use of **JTA Transactions**. You need to define the persistence property **javax.jdo.option.TransactionType** setting it to *JTA*. Then you make use of JTA (or JDO) to demarcate the transactions. So you could do something like

```
UserTransaction ut = (UserTransaction)new InitialContext().lookup
("java:comp/UserTransaction");
PersistenceManager pm = pmf.getPersistenceManager();
try
{
    ut.begin();

    {users code to persist objects}

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

So here we used the JTA API to begin/commit the controlling (*javax.transaction.UserTransaction*).

An alternative is where you don't have a *UserTransaction* started and just use the JDO API, which

will start the *UserTransaction* for you.

```
UserTransaction ut = (UserTransaction)new InitialContext().lookup
("java:comp/UserTransaction");
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin(); // Starts the UserTransaction

    {users code to persist objects}

    tx.commit(); // Commits the UserTransaction
}
finally
{
    pm.close();
}
```

> ❗ You need to set both transactional and nontransactional datasources, and the nontransactional cannot be JTA. The nontransactional is used for schema and sequence operations.

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 DataNucleus JDO 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 "auto-commit" mode.

```
PersistenceManager pm = pmf.getPersistenceManager();

{users code to persist objects}

pm.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 *pm.close()*) and committed with that.

Transaction Isolation

JDO provides a mechanism for specification of the transaction isolation level. This can be specified globally via the persistence property **datanucleus.transaction.isolation** (javax.jdo.option.TransactionIsolationLevel). It accepts the following values

- **read-uncommitted** : dirty reads, non-repeatable reads and phantom reads can occur

- **read-committed** : dirty reads are prevented; non-repeatable reads and phantom reads can occur

- **repeatable-read** : dirty reads and non-repeatable reads are prevented; phantom reads can occur

- **serializable** : dirty reads, non-repeatable reads and phantom reads are prevented

The default (in DataNucleus) is **read-committed**. An attempt to set the isolation level to an unsupported value (for the datastore) will throw a JDOUserException. As an alternative you can also specify it on a per-transaction basis as follows (using the names above).

```
Transaction tx = pm.currentTransaction();
...
tx.setIsolationLevel("read-committed");
```

JDO Transaction Synchronisation

There are situations where you may want to get notified that a transaction is in course of being committed or rolling back. To make that happen, you would do something like

```java
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();

    tx.setSynchronization(new javax.transaction.Synchronization()
    {
        public void beforeCompletion()
        {
            // before commit or rollback
        }

        public void afterCompletion(int status)
        {
            if (status == javax.transaction.Status.STATUS_ROLLEDBACK)
            {
                // rollback
            }
            else if (status == javax.transaction.Status.STATUS_COMMITTED)
            {
                // commit
            }
        }
    });

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

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

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();
    tx.setRollbackOnly();

    {users code to persist objects}

    tx.rollback();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
}
pm.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

```
pm.flush();
```

You can control the flush mode using the persistence property **datanucleus.flush.mode**. This has the following values

- **Auto** : auto-flush changes to the datastore when they are made. This is the default for pessimistic transactions.

- **Manual** : only flush on explicit calls to *pm.flush()* or *tx.commit()*. This is the default for optimistic transactions.

- **Query** : only flush on explicit calls to *pm.flush()* or *tx.commit()*, or just before a Query is executed.

 Extension

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

```
List<ObjectProvider> objs = ((JDOPersistenceManager)pm).getExecutionContext
().getObjectsToBeFlushed();
```

Transaction Savepoints

Extension

ℹ 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.jdo.JDOTransaction;

PersistenceManager pm = pmf.getPersistenceManager();
JDOTransaction tx = (JDOTransaction)pm.currentTransaction();
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();
    }
}
pm.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.

- Lock all records in a datastore and keep them locked until they are ready to commit their changes. These are known as Pessimistic (or datastore) Locking.

- 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.

Pessimistic (Datastore) Locking

Pessimistic locking 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. By default DataNucleus does not currently lock the objects fetched with pessimistic locking, but you can configure this behaviour for RDBMS datastores by setting the persistence property **datanucleus.SerializeRead** to *true*. This will result in all `SELECT ··· FROM ···` statements being changed to be `SELECT ··· FROM ··· FOR UPDATE`. This will be applied only where the underlying RDBMS supports the "FOR UPDATE" syntax. This can be done on a transaction-by-transaction basis by doing

```
Transaction tx = pm.currentTransaction();
tx.setSerializeRead(true);
```

Alternatively, on a per query basis, you would do

```
Query q = pm.newQuery(...);
q.setSerializeRead(true);
```

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).

In terms of the process of pessimistic (datastore) locking, we demonstrate this below.

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

Operation	DataNucleus process	Datastore process
Persist object	Prepare object (3) for persistence	Insert the object (3) into the datastore
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 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. However 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.

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

- Persistence property **datanucleus.flush.mode** when set to *MANUAL* will try to delay all datastore operations until commit/flush.

- Persistence property **datanucleus.datastoreTransactionFlushLimit** represents the number of dirty objects before a flush is performed. This defaults to 1.

Optimistic Locking

Optimistic locking is the other option in JDO. It 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 data is checked just before 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. The user will decide this when generating their MetaData.

Rather than placing version/timestamp columns on all user datastore tables, JDO allows the user to notate particular classes as requiring **optimistic** treatment. This is performed by specifying in MetaData or annotations the details of the field/column to use for storing the version - see versioning. With JDO the version is added in a surrogate column, whereas a vendor extension allows you to have a field in your class ready to store the version.

When the version is stored in a surrogate column in the datastore, JDO provides a helper method for accessing this version. You can call

```
JDOHelper.getVersion(object);
```

and this returns the version as an Object (typically Long or Timestamp). It will return null for a transient object, and will return the version for a persistent object. If the object is not *persistable* then it will also return null.

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	
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**

Here no changes make it to the datastore until the user either commits the transaction, or they invoke flush(). The impact of this is that when performing a query, by default, the results may not contain the modified objects unless they are flushed to the datastore before invoking the query. Depending on whether you need the modified objects to be reflected in the results of the query governs what you do about that. If you invoke flush() just before running the query the query results will include the changes. 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.

Please note that for some datastores (e.g RDBMS) the version check followed by update/delete is performed in a single statement. See also :-

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

Datastore Connections

DataNucleus utilises datastore connections as follows

- PMF : single connection at any one time for datastore-based value generation. Obtained just for the operation, then released

- PMF : single connection at any one time for schema-generation. Obtained just for the operation, then released

- PM : 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).

Datastore connections are obtained from up to 2 connection factories. The primary connection factory is used for persistence operations, and optionally for value generation operations. The secondary connection factory is used for schema generation, and optionally for value generation operations.

> If you are performing any schema generation at runtime then you must define a secondary connection factory.

> If you have multiple threads using the same PersistenceManager 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 JDO 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 *PersistenceManagers*.

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 core 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.

User Connection

JDO defines 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 (for RDBMS)

```java
// Obtain the connection from the JDO implementation
JDOConnection conn = pm.getDataStoreConnection();
try
{
    Object native = conn.getNativeConnection();
    // Cast "native" to the required type for the datastore, see below

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

The "JDOConnection" `Javadoc` in the case of DataNucleus is a wrapper to the native connection for the type of datastore being used. 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 now have a connection allowing direct access to the datastore.

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

Connection Pooling

When you create a *PersistenceManagerFactory* using a connection URL, driver name, and the username/password, this does not 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 when specifying the datastore via the URL. We currently provide support for the following

- RDBMS : Apache DBCP v2, we allow use of externally-defined DBCP2, but also provide a builtin DBCP v2.1.1

- RDBMS : C3P0

- RDBMS : Proxool

- 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*.

So the *PMF* 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*.

So the *PMF* 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 : Proxool

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

So the *PMF* will use connection pooling using Proxool. To do this you will need the `proxool` and `commons-logging` JARs to be in the CLASSPATH.

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

```
datanucleus.connectionPool.maxConnections=10
datanucleus.connectionPool.testSQL=SELECT 1
```

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*.

So the *PMF* 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*.

So the *PMF* 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.maxIdle=5
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*.

So the *PMF* 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, Proxool, 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 PMF
Map properties = new HashMap();
properties.put("datanucleus.ConnectionFactory", ds);

PersistenceManagerFactory pmf = JDOHelper.createPersistenceManagerFactory
("myPersistenceUnit", properties);
```

Note that we haven't passed the *dbUser* and *dbPassword* to the PMF 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 PMF. 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. 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 JavaEE 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 PersistenceManagerFactory, 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);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

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);
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

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
<?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*.

```java
Properties properties = new Properties();
properties.setProperty("javax.persistence.jtaDataSource","java:comp/env/jdbc/datanucleus");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

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("javax.persistence.jtaDataSource","jdbc/datanucleus");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

Using JNDI deployed in component environment:

```
Properties properties = new Properties();
properties.setProperty("javax.persistence.jtaDataSource","java:comp/env/jdbc/datanucle
us");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

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 PMF for each database, and manage use of the appropriate PMF yourself.

- **Separate Schema per Tenant** - as the first option, except use different schemas. In this case you will have a separate PMF for each database schema, and manage use of the appropriate PMF yourself.

- **Same Database/Schema but with a Discriminator in affected Table(s)**. In this case you will have a single PMF, 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.

By default this will add a column **TENANT_ID** to each primary table, of String-based type. You can control this by specifying extension metadata for each persistable class

```xml
<class name="MyClass">
    <extension vendor-name="datanucleus" key="multitenancy-column-name"
value="TENANT"/>
    <extension vendor-name="datanucleus" key="multitenancy-column-length" value=
"24"/>
    ...
</class>
```

or using annotations

```java
@PersistenceCapable
@MultiTenant(column="TENANT", columnLength=24)
public class MyClass
{
    ...
}
```

In all subsequent use of DataNucleus, any "insert" to the primary "table"(s) will also include the TENANT column value. Additionally any query will apply a WHERE clause restricting to a particular value of TENANT column.

If you have enabled multi-tenancy as above but want to disable multitenancy on a class, just specify the following metadata on the class in question

```
<class name="MyClass">
    <extension vendor-name="datanucleus" key="multitenancy-disable" value="true"/>
    ...
</class>
```

or using annotations

```
@PersistenceCapable
@MultiTenant(disable=true)
public class MyClass
{
    ...
}
```

Note that the **Tenant ID** can be set in one of three ways.

- Per PersistenceManagerFactory : just set the persistence property **datanucleus.tenantId** when you start up the PMF, and all access for this PMF will use this Tenant ID

- Per PersistenceManager : set the persistence property **datanucleus.tenantId** when you start up the PMF as the default Tenant ID, and set a property on any PM that you want a different Tenant ID specifying for. Like this

```
PersistenceManager pm = pmf.getPersistenceManager();
... // All operations will apply to default tenant specified in persistence property
for PMF
pm.close();

PersistenceManager pm1 = pmf.getPersistenceManager();
pm1.setProperty("datanucleus.tenantId", "John");
... // All operations will apply to tenant "John"
pm1.close();

PersistenceManager pm2 = pmf.getPersistenceManager();
pm2.setProperty("datanucleus.tenantId", "Chris");
... // All operations will apply to tenant "Chris"
pm2.close();
```

- Per datastore access : When creating the PMF 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?

Bean Validation

Extension

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 JDO (DataNucleus extension) 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 **datanucleus.validation.mode** to one of *auto* (default), *none*, or *callback*

- Optionally set the persistence property(s) **datanucleus.validation.group.pre-persist**, **datanucleus.validation.group.pre-update**, **datanucleus.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 JDO as you normally would for persisting objects

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

```
@PersistenceCapable
public class Person
{
    @PrimaryKey
    @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 `@PersistenceCapable`.

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 JDO "length" and "allowsNull" since they equate to the same thing. This is enabled via the persistence property **datanucleus.metadata.javaxValidationShortcuts**.

Fetch Groups

When an object is retrieved from the datastore by JDO 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 a **fetch group**. There are 3 types of "fetch groups" to consider

- Default Fetch Group : defined in all JDO specs, containing the fields of a class that will be retrieved by default (with no user specification).

- Named Fetch Groups : defined by the JDO specification, and defined in MetaData (XML/annotations) with the fields of a class that are part of that fetch group. The definition here is *static*

- Dynamic Fetch Groups : Programmatic definition of fetch groups at runtime via an API

The **fetch group** in use for a class is controled via the *FetchPlan* `Javadoc` interface. To get a handle on the current *FetchPlan* we do

```
FetchPlan fp = pm.getFetchPlan();
```

Default Fetch Group

JDO provides an initial fetch group, comprising the fields that will be retrieved when an object is retrieved if the user does nothing to define the required behaviour. By default the *default fetch group* comprises all fields of the following types :-

- primitives : boolean, byte, char, double, float, int, long, short

- Object wrappers of primitives : Boolean, Byte, Character, Double, Float, Integer, Long, Short

- java.lang.String, java.lang.Number, java.lang.Enum

- java.math.BigDecimal, java.math.BigInteger

- java.util.Date

If you wish to change the **Default Fetch Group** for a class you can update the Meta-Data for the class as follows

```
@Persistent(defaultFetchGroup="true")
SomeType fieldX;
```

or using XML metadata

```
<class name="MyClass">
    ...
    <field name="fieldX" default-fetch-group="true"/>
</class>
```

When a *PersistenceManager* is created it starts with a FetchPlan of the "default" fetch group. That is, if we call

```
Collection fetchGroups = fp.getGroups();
```

this will have one group, called "default". At runtime, if you have been using other fetch groups and want to revert back to the default fetch group at any time you simply do

```
fp.setGroup(FetchPlan.DEFAULT);
```

Named Fetch Groups

As mentioned above, JDO allows specification of users own fetch groups. These are specified in the MetaData of the class. For example, if we have the following class

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

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

```
@PersistenceCapable
@FetchGroup(name="otherfield", members={@Persistent(name="other")})
public class MyClass
{
    ...
}
```

or using XML metadata

```xml
<package name="mydomain">
    <class name="MyClass">
        <field name="name">
            <column length="100" jdbc-type="VARCHAR"/>
        </field>
        <field name="coll" persistence-modifier="persistent">
            <collection element-type="mydomain.Address"/>
            <join/>
        </field>
        <field name="other" persistence-modifier="persistent"/>
        <fetch-group name="otherfield">
            <field name="other"/>
        </fetch-group>
    </class>
</package>
```

So we have defined a fetch group called "otherfield" that just includes the field with name *other*. We can then use this at runtime in our persistence code.

```java
PersistenceManager pm = pmf.getPersistenceManager();
pm.getFetchPlan().addGroup("otherfield");

... (load MyClass object)
```

By default the *FetchPlan* will include the default fetch group. We have changed this above by **adding** the fetch group "otherfield", so when we retrieve an object using this *PersistenceManager* we will be retrieving the fields *name* AND *other* since they are both in the current *FetchPlan*. We can take the above much further than what is shown by defining nested fetch groups in the MetaData. In addition we can change the *FetchPlan* just before any *PersistenceManager* operation to control what is fetched during that operation. The user has full flexibility to add many groups to the current **Fetch Plan**. This gives much power and control over what will be loaded and when. A big improvement over the "default" fetch group.

The *FetchPlan* applies not just to calls to *PersistenceManager.getObjectById()*, but also to *PersistenceManager.newQuery()*, *PersistenceManager.getExtent()*, *PersistenceManager.detachCopy* and much more besides.

Dynamic Fetch Groups

The mechanism above provides static fetch groups defined in XML or annotations. That is great when you know in advance what fields you want to fetch. In some situations you may want to define your fields to fetch at run time. It operates as follows

```
import org.datanucleus.FetchGroup;

// Create a FetchGroup on the PMF called "TestGroup" for MyClass
FetchGroup grp = myPMF.getFetchGroup(MyClass.class, "TestGroup");
grp.addMember("field1").addMember("field2");

// Add this group to the fetch plan (using its name)
fp.addGroup("TestGroup");
```

So we use the DataNucleus PMF as a way of creating a FetchGroup, and then register that
FetchGroup with the PMF for use by all PMs. We then enable our FetchGroup for use in the
FetchPlan by using its group name (as we do for a static group). The FetchGroup allows you to
add/remove the fields necessary so you have full API control over the fields to be fetched.

Fetch Depth

The basic fetch group defines which fields are to be fetched. It doesn't explicitly define how far
down an object graph is to be fetched. JDO provides two ways of controlling this.

The first is to set the **maxFetchDepth** for the *FetchPlan*. This value specifies how far out from the
root object the related objects will be fetched. A positive value means that this number of
relationships will be traversed from the root object. A value of -1 means that no limit will be placed
on the fetching traversal. The default is 1. Let's take an example

```
public class MyClass1
{
    MyClass2 field1;
    ...
}

public class MyClass2
{
    MyClass3 field2;
    ...
}

public class MyClass3
{
    MyClass4 field3;
    ...
}
```

and we want to detach *field1* of instances of *MyClass1*, down 2 levels - so detaching the initial
"field1" *MyClass2* object, and its "field2" *MyClass3* instance. So we define our fetch-groups like this

```xml
<class name="MyClass1">
    ...
    <fetch-group name="includingField1">
        <field name="field1"/>
    </fetch-group>
</class>
<class name="MyClass2">
    ...
    <fetch-group name="includingField2">
        <field name="field2"/>
    </fetch-group>
</class>
```

and we then define the **maxFetchDepth** as 2, like this

```java
pm.getFetchPlan().setMaxFetchDepth(2);
```

A further refinement to this global fetch depth setting is to control the fetching of recursive fields. This is performed via a MetaData setting "recursion-depth". A value of 1 means that only 1 level of objects will be fetched. A value of -1 means there is no limit on the amount of recursion. The default is 1. Let's take an example

```java
public class Directory
{
    Collection children;
    ...
}
```

```xml
<class name="Directory">
    <field name="children">
        <collection element-type="Directory"/>
    </field>

    <fetch-group name="grandchildren">
        <field name="children" recursion-depth="2"/>
    </fetch-group>
    ...
</class>
```

So when we fetch a Directory, it will fetch 2 levels of the *children* field, hence fetching the children and the grandchildren.

Fetch Size

A FetchPlan can also be used for defining the fetching policy when using queries. This can be set

using

```
pm.getFetchPlan().setFetchSize(value);
```

The default is *FetchPlan.FETCH_SIZE_OPTIMAL* which leaves it to DataNucleus to optimise the fetching of instances. A positive value defines the number of instances to be fetched. Using *FetchPlan.FETCH_SIZE_GREEDY* means that all instances will be fetched immediately.

Lifecycle Callbacks

JDO defines a mechanism whereby a persistable class can be marked as a listener for lifecycle events. Alternatively a separate 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

Instance Callbacks

JDO defines an interface for persistable classes so that they can be notified of events in their own lifecycle and perform any additional operations that are needed at these checkpoints. This is a complement to the Lifecycle Listeners interface which provides listeners for all objects of particular classes, with the events sent to a listener. With **InstanceCallbacks** the *persistable* class is the destination of the lifecycle events. As a result the **Instance Callbacks** method is more intrusive than the method of *Lifecycle Listeners* in that it requires methods adding to each class that wishes to receive the callbacks.

The **InstanceCallbacks** interface is documented here.

To give an example of this capability, let us define a class that needs to perform some operation just before it's object is deleted.

```java
public class MyClass implements InstanceCallbacks
{
    String name;

    ... (class methods)


    public void jdoPostLoad() {}
    public void jdoPreClear() {}
    public void jdoPreStore() {}

    public void jdoPreDelete()
    {
        // Perform some operation just before being deleted.
    }
}
```

We have implemented *InstanceCallbacks* and have defined the 4 required methods. Only one of these is of importance in this example.

These methods will be called just before storage in the data store (*jdoPreStore*), just before clearing (*jdoPreClear*), just after being loaded from the datastore (*jdoPostLoad*) and just before being deleted (*jdoPreDelete*).

JDO also has 2 additional callbacks to complement *InstanceCallbacks*. These are *AttachCallback* Javadoc and *DetachCallback* Javadoc. If you want to intercept attach/detach events your class can

implement these interfaces. You will then need to implement the following methods

```
public interface AttachCallback
{
    public void jdoPreAttach();
    public void jdoPostAttach(Object attached);
}

public interface DetachCallback
{
    public void jdoPreDetach();
    public void jdoPostDetach(Object detached);
}
```

Lifecycle Listeners

JDO defines an interface for the PersistenceManager and PersistenceManagerFactory whereby a user can register a listener for persistence events. The user provides a listener for either all classes, or a set of defined classes, and the JDO implementation calls methods on the listener when the required events occur. This provides the user application with the power to monitor the persistence process and, where necessary, append related behaviour. Specifying the listeners on the PersistenceManagerFactory has the benefits that these listeners will be added to all PersistenceManagers created by that factory, and so is for convenience really. This facility is a complement to the Instance Callbacks facility which allows interception of events on an instance by instance basis. The **Lifecycle Listener** process is much less intrusive than the process provided by *Instance Callbacks*, allowing a class external to the persistence process to perform the listening.

The **InstanceLifecycleListener** interface is documented here.

To give an example of this capability, let us define a Listener for our persistence process.

```java
public class LoggingLifecycleListener implements CreateLifecycleListener,
    DeleteLifecycleListener, LoadLifecycleListener, StoreLifecycleListener
{
    public void postCreate(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : create for " +
            ((Persistable)event.getSource()).dnGetObjectId());
    }

    public void preDelete(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : preDelete for " +
            ((Persistable)event.getSource()).dnGetObjectId());
    }

    public void postDelete(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : postDelete for " +
            ((Persistable)event.getSource()).dnGetObjectId());
    }

    public void postLoad(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : load for " +
            ((Persistable)event.getSource()).dnGetObjectId());
    }

    public void preStore(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : preStore for " +
            ((Persistable)event.getSource()).dnGetObjectId());
    }

    public void postStore(InstanceLifecycleEvent event)
    {
        log.info("Lifecycle : postStore for " +
            ((Persistable)event.getSource()).dnGetObjectId());
    }
}
```

Here we've provided a listener to receive events for CREATE, DELETE, LOAD, and STORE of objects. These are the main event types and in our simple case above we will simply log the event. All that remains is for us to register this listener with the PersistenceManager, or PersistenceManagerFactory

```java
pm.addInstanceLifecycleListener(new LoggingLifecycleListener(), null);
```

When using this interface the user should always remember that the listener is called within the

same transaction as the operation being reported and so any changes they then make to the objects in question will be reflected in that objects state.

Register the listener with the PersistenceManager or PersistenceManagerFactory provide different effects. Registering with the PersistenceManagerFactory means that all PersistenceManagers created by it will have the listeners registered on the PersistenceManagerFactory called. Registering the listener with the PersistenceManager will only have the listener called only on events raised only by the PersistenceManager instance.
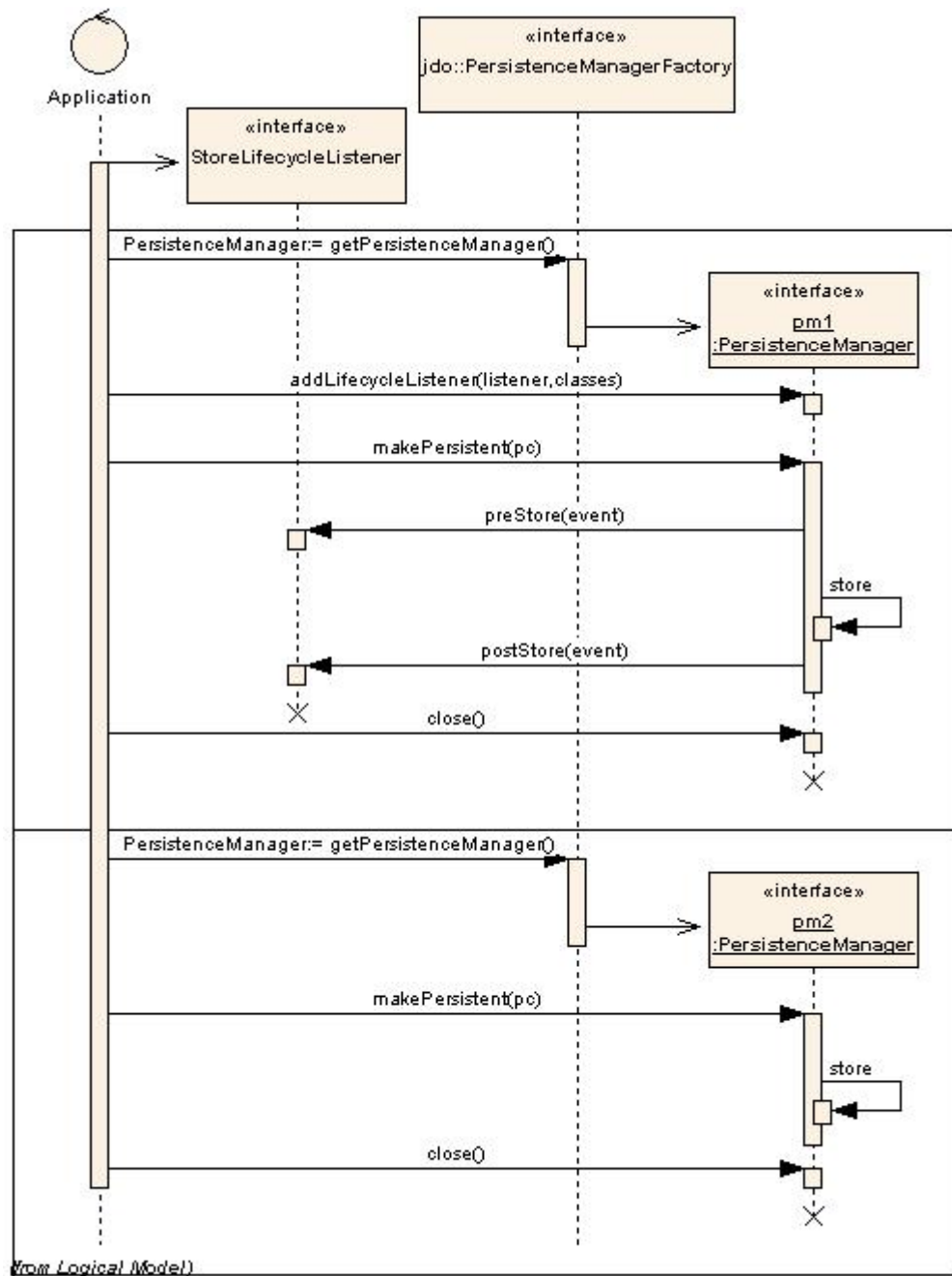
The above diagram displays the sequence of actions for a listener registered only in the PersistenceManager. Note that a second PersistenceManager will not make calls to the listener registered in the first PersistenceManager.
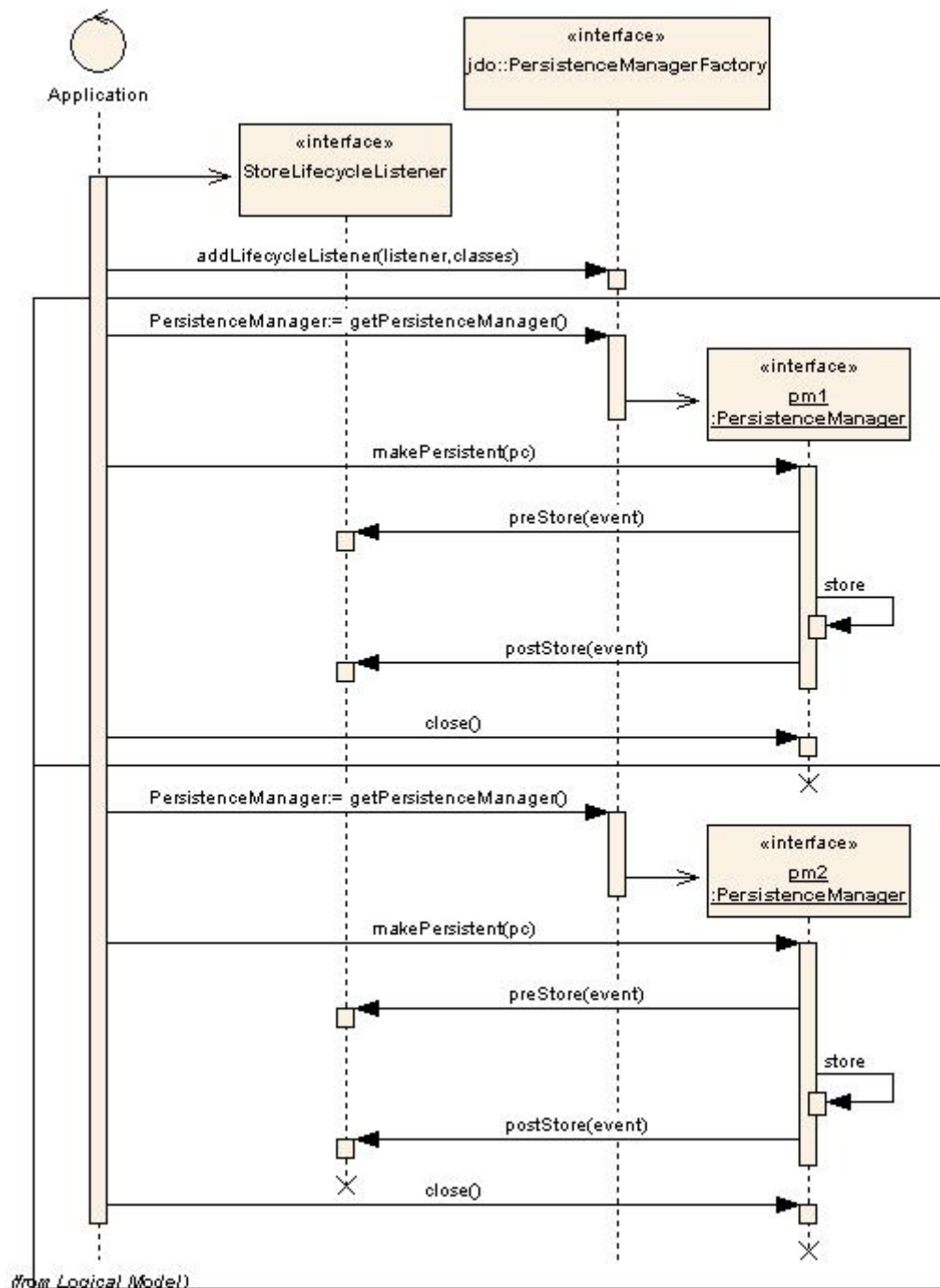
The above diagram displays the sequence of actions for a listener registered in the PersistenceManagerFactory. All events raised in a PersistenceManager obtained from the PersistenceManagerFactory will make calls to the listener registered in the PersistenceManagerFactory.

DataNucleus supports the following instance lifecycle listener types

- **AttachLifecycleListener** - all attach events
- **ClearLifecycleListener** - all clear events
- **CreateLifecycelListener** - all object create events
- **DeleteLifecycleListener** - all object delete events
- **DetachLifecycleListener** - all detach events

- **DirtyLifecycleListener** - all dirty events

- **LoadLifecycleListener** - all load events

- **StoreLifecycleListener** - all store events

The default JDO lifecycle listener *StoreLifecycleListener* only informs the listener of the object being stored. It doesn't provide information about the fields being stored in that event. DataNucleus extends the JDO specification and on the "preStore" event it will return an instance of *org.datanucleus.api.jdo.FieldInstanceLifecycleEvent* (which extends the JDO InstanceLifecycleEvent) and provides access to the names of the fields being stored.

```
public class FieldInstanceLifecycleEvent extends InstanceLifecycleEvent
{
    ...

    /**
     * Accessor for the field names affected by this event
     * @return The field names
     */
    public String[] getFieldNames()
    ...
}
```

If the store event is the persistence of the object then this will return all field names. If instead just particular fields are being stored then you just receive those fields in the event. So the only thing to do to utilise this DataNucleus extension is cast the received event to *org.datanucleus.FieldInstanceLifecycleEvent*

JavaEE Environments

The JavaEE framework is widely used, providing a container within which java processes operate and it provides mechanisms for, amongst other things, transactions (JTA), and for connecting to other (3rd party) utilities (using Java Connector Architecture, JCA). DataNucleus Access Platform can be utilised within a JavaEE environment either in the same way as you use it for JavaSE, or via this JCA system, and we provide a Resource Adaptor (RAR file) containing this JCA adaptor allowing Access Platform to be used with the likes of WebLogic and JBoss. Instructions are provided for the following JavaEE servers

- WebLogic
- JBoss 3.0/3.2
- JBoss 4.0
- JBoss 7.0
- Jonas 4.8

> The main thing to mention here is that you can use DataNucleus in a JavaEE environment just like you use any other library, following the documentation for JavaSE. Consequently you do not *need* the JCA Adaptor for this usage. You solely use the JCA Adaptor if you want to fully integrate with JavaEE; by this we mean make use of transaction demarcation (and so avoid having to put tx.begin/commit).

The remainder of these notes refer to JCA usage. The provided DataNucleus JCA rar provides default resource adapter descriptors, one general, and the other for the WebLogic JavaEE server. These resource adapter descriptors can be configured to meet your needs, for example allowing XA transactions instead of the default Local transactions.

Requirements

To use DataNucleus with JCA the first thing that you will require is the `datanucleus-jdo-jca-5.1.rar` file (available from the downloads).

DataNucleus Resource Adaptor and transactions

A great advantage of DataNucleus implementing the ManagedConnection interface is that the JavaEE container manages transactions for you (no need to call the begin/commit/rollback-methods). Currently, local transactions and distributed (XA) transactions are supported. Within a JavaEE environment, JDO transactions are nested in JavaEE transactions. All you have to do is to declare that a method needs transaction management. This is done in the EJB meta data. Here you will see, how a SessionBean implementation could look like.

The EJB meta data is defined in a file called ejb-jar.xml and can be found in the META-INF directory of the jar you deploy. Suppose you deploy a bean called DataNucleusBean, your ejb-jar.xml should contain the following configuration elements:

```
<session>
    <ejb-name>DataNucleusBean</ejb-name>
    ...
    <transaction-type>Container</transaction-type>
    ...
<session>
```

Imagine your bean defines a method called testDataNucleusTrans():

```
<container-transaction>
    <method >
        <ejb-name>DataNucleusBean</ejb-name>
        ...
        <method-name>testDataNucleusTrans</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
```

You hereby define that transaction management is required for this method. The container will automatically begin a transaction for this method. It will be commited if no error occurs or rolled back otherwise. A potential SessionBean implementation containing methods to retrieve a PersistenceManager then could look like this:

```
public abstract class DataNucleusBean implements SessionBean
{
    // EJB methods
    public void ejbCreate()
    throws CreateException
    {
    }

    public void ejbRemove()
    throws EJBException, RemoteException
    {
    }

    // convenience methods to get
    // a PersistenceManager

    /**
     * static final for the JNDI name of the PersistenceManagerFactory
     */
    private static final String PMF_JNDI_NAME = "java:/datanucleus1";

    /**
     * Method to get the current InitialContext
     */
    private InitialContext getInitialContext() throws NamingException
```

```
    {
        InitialContext initialContext = new InitialContext();
        // or other code to create the InitialContext eg. new
InitialContext(myProperies);
        return initialContext;
    }


    /**
     * Method to lookup the PersistenceManagerFactory
     */
    private PersistenceManagerFactory getPersitenceManagerFactory(InitialContext
context)
    throws NamingException
    {
        return (PersistenceManagerFactory) context.lookup(PMF_JNDI_NAME);
    }


    /**
     * Method to get a PersistenceManager
     */
    public PersistenceManager getPersistenceManager()
    throws NamingException
    {
        return getPersitenceManagerFactory(getInitialContext()).
getPersistenceManager();
    }


    // Now finally the bean method within a transaction
    public void testDataNucleusTrans()
    throws Exception
    {
        PersistenceManager pm = getPersistenceManager()
        try
        {
            // Do something with your PersistenceManager
        }
        finally
        {
            // close the PersistenceManager
            pm.close();
        }
    }
}
```

Make sure that you close the PersistenceManager in your bean methods. If you don't, the JavaEE server will usually close it for you (one of the advantages), but of course not without a warning or error message.

These instructions were adapted from a contribution by a DataNucleus user Alexander Bieber

Persistence Properties

When creating a PMF using the JCA adaptor, you should specify your persistence properties using a persistence.xml or jdoconfig.xml. This is because DataNucleus JCA adapter from version 1.2.2 does not support Java bean setters/getters for all properties - since it is an inefficient and inflexible mechanism for property specification. The more recent `persistence.xml` and `jdoconfig.xml` methods lead to more extensible code.

General configuration

A resource adapter has one central configuration file `/META-INF/ra.xml` which is located within the rar file and which defines the default values for all instances of the resource adapter (i.e. all instances of *PersistenceManagerFactory*). Additionally, it uses one or more deployment descriptor files (in JBoss, for example, they are named `*-ds.xml`) to set up the instances. In these files you can override the default values from the `ra.xml`.

Since it is bad practice (and inconvenient) to edit a library's archive (in this case the `datanucleus-jdo-jca-$5.1.rar`) for changing the configuration (it makes updates more complicated, for example), it is recommended, not to edit the `ra.xml` within DataNucleus' rar file, but instead put all your configuration into your deployment descriptors. This way, you have a clean separation of which files you maintain (your deployment descriptors) and which files are maintained by others (the libraries you use and which you simply replace in case of an update).

Nevertheless, you might prefer to declare default values in the `ra.xml` in certain circumstances, so here's an example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connector PUBLIC "-//Sun Microsystems, Inc.//DTD Connector 1.0//EN"
    "http://java.sun.com/dtd/connector_1_0.dtd">
<connector>
    <display-name>DataNucleus Connector</display-name>
    <description></description>
    <vendor-name>DataNucleus Team</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>JDO Adaptor</eis-type>
    <version>1.0</version>
    <resourceadapter>
        <managedconnectionfactory-
class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnectionfac
tory-class>
        <connectionfactory-interface>
javax.resource.cci.ConnectionFactory</connectionfactory-interface>
        <connectionfactory-impl-
class>org.datanucleus.jdo.connector.PersistenceManagerFactoryImpl</connectionfactory-
impl-class>
        <connection-interface>javax.resource.cci.Connection</connection-interface>
        <connection-impl-class>
org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
        <transaction-support>LocalTransaction</transaction-support>
        <config-property>
          <config-property-name>ConnectionFactoryName</config-property-name>
          <config-property-type>java.lang.String</config-property-type>
          <config-property-value>jdbc/ds</config-property-value>
        </config-property>
        <authentication-mechanism>
          <authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
          <credential-interface>
javax.resource.security.PasswordCredential</credential-interface>
        </authentication-mechanism>
        <reauthentication-support>false</reauthentication-support>
    </resourceadapter>
</connector>
```

To define persistence properties you should make use of `persistence.xml` or `jdoconfig.xml` and refer to the documentation for [persistence properties](#) for full details of the properties.

WebLogic

To use DataNucleus on Weblogic the first thing that you will require is the `datanucleus-jdo-jca-5.1.rar` file. You then may need to edit the `/META-INF/weblogic-ra.xml` file to suit the exact version of your WebLogic server (the included file is for WebLogic 8.1).

You then deploy the RAR file on your WebLogic server.

JBoss 3.0/3.2

To use DataNucleus on JBoss (Ver 3.2) the first thing that you will require is the `datanucleus-jdo-jca-5.1.rar` file. You should put this in the *deploy ("${JBOSS}/server/default/deploy/")* directory of your JBoss installation.

You then create a file, also in the *deploy* directory with name `datanucleus-ds.xml`. To give a guide on what this file will typically include, see the following

```xml
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
    <tx-connection-factory>
        <jndi-name>datanucleus</jndi-name>
        <adapter-display-name>DataNucleus Connector</adapter-display-name>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.mysql.jdbc.Driver</config-property>
        <config-property name="ConnectionURL"
            type="java.lang.String">jdbc:mysql://localhost/yourdbname</config-property>
        <config-property name="UserName"
            type="java.lang.String">yourusername</config-property>
        <config-property name="Password"
            type="java.lang.String">yourpassword</config-property>
    </tx-connection-factory>

    <tx-connection-factory>
        <jndi-name>datanucleus1</jndi-name>
        <adapter-display-name>DataNucleus Connector</adapter-display-name>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.mysql.jdbc.Driver</config-property>
        <config-property name="ConnectionURL"
            type="java.lang.String">jdbc:mysql://localhost/yourdbname1</config-property>
        <config-property name="UserName"
            type="java.lang.String">yourusername</config-property>
        <config-property name="Password"
            type="java.lang.String">yourpassword</config-property>
    </tx-connection-factory>

    <tx-connection-factory>
        <jndi-name>datanucleus2</jndi-name>
        <adapter-display-name>DataNucleus Connector</adapter-display-name>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.mysql.jdbc.Driver</config-property>
        <config-property name="ConnectionURL"
            type="java.lang.String">jdbc:mysql://localhost/yourdbname2</config-property>
        <config-property name="UserName"
            type="java.lang.String">yourusername</config-property>
        <config-property name="Password"
            type="java.lang.String">yourpassword</config-property>
    </tx-connection-factory>
</connection-factories>
```

This example creates 3 connection factories to MySQL databases, but you can create as many or as few as you require for your system to whichever databases you prefer (as long as they are supported by DataNucleus). With the above definition we can then use the JNDI names *java:/datanucleus, java:/datanucleus1*, and *java:/datanucleus2* to refer to our datastores.

Note, that you can use separate deployment descriptor files. That means, you could for example create the three files `datanucleus1-ds.xml`, `datanucleus2-ds.xml` and `datanucleus3-ds.xml` with each declaring one *PersistenceManagerFactory* instance. This is useful (or even required) if you need a distributed configuration. In this case, you can use JBoss' hot deployment feature and deploy a new *PersistenceManagerFactory*, while the server is running (and working with the existing PMFs): If you create a new `*-ds.xml` file (instead of modifying an existing one), the server does not undeploy anything (and thus not interrupt ongoing work), but will only add the new connection factory to the JNDI.

You are now set to work on DataNucleus-enabling your actual application. As we have said, you can use the above JNDI names to refer to the datastores, so you could do something like the following to access the PersistenceManagerFactory to one of your databases.

```
import javax.jdo.PersistenceManagerFactory;

InitialContext context = new InitialContext();
PersistenceManagerFactory pmf = (PersistenceManagerFactory)context.lookup
("java:/datanucleus1");
```

These instructions were adapted from a contribution by a DataNucleus user Marco Schulze.

JBoss 4.0

With JBoss 4.0 there are some changes in configuration relative to JBoss 3.2 in order to allow use some new features of JCA 1.5. Here you will see how to configure JBoss 4.0 to use with DataNucleus JCA adapter for DB2.

To use DataNucleus on JBoss 4.0 the first thing that you will require is the `datanucleus-jdo-jca-5.1.rar` file. You should put this in the deploy directory ("${JBOSS}/server/default/deploy/") of your JBoss installation. Additionally, you have to remember to put any JDBC driver files to lib directory ("${JBOSS}/server/default/lib/") if JBoss does not have them installed by default. In case of DB2 you need to copy `db2jcc.jar` and `db2jcc_license_c.jar`.

You then create a file, also in the deploy directory with name `datanucleus-ds.xml`. To give a guide on what this file will typically include, see the following

```xml
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
    <tx-connection-factory>
        <jndi-name>datanucleus</jndi-name>
        <rar-name>datanucleus-jca-version}.rar</rar-name> <!-- the name here must be
the same as JCA adapter filename -->
        <connection-definition>javax.resource.cci.ConnectionFactory</connection-
definition>
        <config-property name="ConnectionDriverName"
            type="java.lang.String">com.ibm.db2.jcc.DB2Driver</config-property>
        <config-property name="ConnectionURL"
            type="java.lang.String"
>jdbc:derby:net://localhost:1527/"directory_of_your_db_files"</config-property>
        <config-property name="UserName"
            type="java.lang.String">app</config-property>
        <config-property name="Password"
            type="java.lang.String">app</config-property>
    </tx-connection-factory>
</connection-factories>
```

You are now set to work on DataNucleus-enabling your actual application. You can use the above JNDI name to refer to the datastores, and so you could do something like the following to access the PersistenceManagerFactory to one of your databases.

```java
import javax.jdo.PersistenceManagerFactory;

InitialContext context=new InitialContext();
PersistenceManagerFactory pmFactory=(PersistenceManagerFactory)context.lookup
("java:/datanucleus");
```

These instructions were adapted from a contribution by a DataNucleus user Maciej Wegorkiewicz

JBoss 7.0

A tutorial for running DataNucleus under JBoss 7 is available on the internet, provided by a DataNucleus user Kiran Kumar.

Jonas

To use DataNucleus on Jonas the first thing that you will require is the `datanucleus-jdo-jca-5.1.rar` file. You then may need to edit the `/META-INF/jonas-ra.xml` file to suit the exact version of your Jonas server (the included file is tested for Jonas 4.8).

You then deploy the RAR file on your Jonas server.

Transaction Support

DataNucleus JCA adapter supports both Local and XA transaction types. Local means that a transaction will not have more than one resource managed by a Transaction Manager and XA means that multiple resources are managed by the Transaction Manager. Use XA transaction if DataNucleus is configured to use data sources deployed in application servers, or if other resources such as JMS connections are used in the same transaction, otherwise use Local transaction.

You need to configure the *ra.xml* file with the appropriate transaction support, which is either *XATransaction* or *LocalTransaction*. See the example:

```
<connector>
    <display-name>DataNucleus Connector</display-name>
    <description></description>
    <vendor-name>DataNucleus Team</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>JDO Adaptor</eis-type>
    <version>1.0</version>
    <resourceadapter>
        <managedconnectionfactory-
class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnectionfac
tory-class>
        <connectionfactory-interface>
javax.resource.cci.ConnectionFactory</connectionfactory-interface>
        <connectionfactory-impl-
class>org.datanucleus.jdo.connector.PersistenceManagerFactoryImpl</connectionfactory-
impl-class>
        <connection-interface>javax.resource.cci.Connection</connection-interface>
        <connection-impl-class>
org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
        <transaction-support>XATransaction</transaction-support> <!-- change this line
-->
    ...
```

Data Source

To use a data source, you have to configure the connection factory name in `ra.xml` file. See the example:

```
<connector>
    <display-name>DataNucleus Connector</display-name>
    <description></description>
    <vendor-name>DataNucleus Team</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>JDO Adaptor</eis-type>
    <version>1.0</version>
    <resourceadapter>
        <managedconnectionfactory-
class>org.datanucleus.jdo.connector.ManagedConnectionFactoryImpl</managedconnectionfac
tory-class>
        <connectionfactory-interface>
javax.resource.cci.ConnectionFactory</connectionfactory-interface>
        <connectionfactory-impl-
class>org.datanucleus.jdo.connector.PersistenceManagerFactoryImpl</connectionfactory-
impl-class>
        <connection-interface>javax.resource.cci.Connection</connection-interface>
        <connection-impl-class>
org.datanucleus.jdo.connector.PersistenceManagerImpl</connection-impl-class>
        <transaction-support>XATransaction</transaction-support>

        <config-property>
            <config-property-name>ConnectionFactoryName</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>jndiName_for_datasource_1</config-property-value>
        </config-property>
            <config-property>
            <config-property-name>ConnectionResourceType</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>JTA</config-property-value>
        </config-property>
        <config-property>
            <config-property-name>ConnectionFactory2Name</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>jndiName_for_datasource_2</config-property-value>
        </config-property>
    ...
```

See also :

- (RDBMS) Data Sources usage with DataNucleus

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. Please refer to the following guide(s) for assistance until a definitive guide can be provided

- Guide to use of DataNucleus with OSGi and Spring dmServer

- Guide to DataNucleus inside Eclipse RCP

- Guide to DataNucleus with Spring and Eclipse RCP

- Guide to using Log4J with DataNucleus under OSGi

Some key points around integration with OSGi are as follows :-

- 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 `javax.jdo.jar` that is included in the DataNucleus distribution is OSGi-enabled.

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

- When using DataNucleus in an OSGi environments set the persistence property **datanucleus.plugin.pluginRegistryClassName** to *org.datanucleus.plugin.OSGiPluginRegistry*

- If you redeploy a JDO-enabled OSGi application, likely you will need to *refresh* the javax.jdo and maybe other bundles.

Please make use of the OSGi sample for JDO in case it is of use. Use of OSGi is notorious for class loading oddities, so it may be necessary to refine this sample for your situation. We welcome any feedback to improve it.

HOWTO Use Datanucleus with OSGi and Spring DM

This guide was written by Jasper Siepkes.

This guide is based on my personal experience and is not the authoritative guide to using DataNucleus with OSGi and Spring DM. I've updated this guide to use DataNucleus 3.x and Eclipse Gemini (formerly Spring DM). I haven't extensively tested it yet. This guide explains how to use DataNucleus, Spring, OSGi and the OSGi blueprint specification together. This guide assumes the reader is familiar with concepts like OSGi, Spring, JDO, DataNucleus etc. This guide only explains how to wire these technologies together and not how they work. Now there have been a lot of (name) changes in over a short course of time. Some webpages might not have been updated yet so to undo some of the confusion created here is the deal with Eclipse Gemini. Eclipse Gemini started out as Spring OSGi, which was later renamed to Spring Dynamic Modules or Spring DM for short. Spring DM is *NOT* to be confused with Spring DM Server. Spring DM Server is a complete server product with management UI and tons of other features. Spring DM is the core of Spring DM Server and provides only the service / dependency injection part. At some point in time the Spring team

decided to donate their OSGi efforts to the Eclipse foundation. Spring DM became Eclipse Gemini and Spring DM Server became Eclipse Virgo. The whole Spring OSGi / Spring DM / Eclipse Gemini later became standardised as the OSGi Blueprint specification. To summarise: Spring OSGi = Spring DM = Eclipse Gemini, Spring DM Server = Eclipse Virgo.

Technologies used in this guide are:

- IDE (Eclipse 3.7)

- OSGi (Equinox 3.7.1)

- JDO (DataNucleus 3.x)

- Dependency Injection (Spring 3.0.6)

- OSGi Blueprint (Eclipse Gemini BluePrint 1.0.0)

- Datastore (PostgreSQL 8.3, altough any datastore supported by DataNucleus can be used)

We are going to start by creating a clean OSGi target platform. Start by creating an empty directory which is going to house all the bundles for our target platform.

Step 1 : Adding OSGi

The first ingredient we are adding to our platform is the OSGi implementation. In this guide we will use Eclipse Equinox as our OSGi implementation. However one could also use Apache Felix, Knoplerfish, Concierge or any other compatible OSGi implementation for this purpose. Download the `org.eclipse.osgi_3.7.1.R37x_v20110808-1106.jar` ("Framework Only" download) from the Eclipse Equinox website and put in the target platform.

Step 2 - Adding DI

We are now going to add the Spring, Spring ORM, Spring JDBC, Spring Transaction and Spring DM bundles to our target platform. Download the Spring Community distribution from their website (`spring-framework-3.0.6.RELEASE.zip`). Extract the following files to our target platform directory:

- `org.springframework.aop-3.0.6.RELEASE.jar`
- `org.springframework.asm-3.0.6.RELEASE.jar`
- `org.springframework.aspects-3.0.6.RELEASE.jar`
- `org.springframework.beans-3.0.6.RELEASE.jar`
- `org.springframework.context.support-3.0.6.RELEASE.jar`
- `org.springframework.context-3.0.6.RELEASE.jar`
- `org.springframework.core-3.0.6.RELEASE.jar`
- `org.springframework.expression-3.0.6.RELEASE.jar`
- `org.springframework.jdbc-3.0.6.RELEASE.jar`
- `org.springframework.orm-3.0.6.RELEASE.jar`
- `org.springframework.spring-library-3.0.6.RELEASE.libd`
- `org.springframework.transaction-3.0.6.RELEASE.jar`

Step 3 - Adding OSGi Blueprint

Download the Eclipse Gemini release from their website (`gemini-blueprint-1.0.0.RELEASE.zip`) and extract the following files to our target platform:

- `gemini-blueprint-core-1.0.0.RELEASE.jar`
- `gemini-blueprint-extender-1.0.0.RELEASE.jar`
- `gemini-blueprint-io-1.0.0.RELEASE.jar`

Step 4 - Adding ORM

We are now going to add JDO and DataNucleus to our target platform.

- `datanucleus-core-XXX.jar`
- `datanucleus-api-jdo-XXX.jar`
- `datanucleus-rdbms-XXX.jar`
- `javax.jdo-3.2.0-m5.jar`

Step 5 - Adding miscellaneous bundles

The following bundles are dependencies of our core bundles and can be downloaded from the Spring Enterprise Bundle Repository

- `com.springsource.org.aopalliance-1.0.0.jar` (Dependency of Spring AOP, the core AOP bundle.)
- `com.springsource.org.apache.commons.logging-1.1.1.jar` (Dependency of various Spring bundles, logging abstraction library.)
- `com.springsource.org.postgresql.jdbc4-8.3.604.jar` (PostgreSQL JDBC driver, somewhat dated.)

We now have a basic target platform. This is how the directory housing the target platform looks on my PC:

```
$ ls -las
    4 drwxrwxr-x 2 siepkes siepkes    4096 Oct 22 15:28 .
    4 drwxrwxr-x 3 siepkes siepkes    4096 Oct 22 15:29 ..
    8 -rw-r----- 1 siepkes siepkes    4615 Oct 22 15:27
com.springsource.org.aopalliance-1.0.0.jar
   68 -rw-r----- 1 siepkes siepkes   61464 Oct 22 15:28
com.springsource.org.apache.commons.logging-1.1.1.jar
  472 -rw-r----- 1 siepkes siepkes  476053 Oct 22 15:28
com.springsource.org.postgresql.jdbc4-8.3.604.jar
  312 -rw-r----- 1 siepkes siepkes  314358 Oct  2 11:36 datanucleus-api-jdo-5.0.1.jar
 1624 -rw-r----- 1 siepkes siepkes 1658797 Oct  2 11:36 datanucleus-core-5.0.1.jar
 1400 -rw-r----- 1 siepkes siepkes 1427439 Oct  2 11:36 datanucleus-rdbms-5.0.1.jar
  572 -rw-r----- 1 siepkes siepkes  578205 Aug 22 22:37 gemini-blueprint-core-
1.0.0.RELEASE.jar
  180 -rw-r----- 1 siepkes siepkes  178525 Aug 22 22:37 gemini-blueprint-extender-
1.0.0.RELEASE.jar
   32 -rw-r----- 1 siepkes siepkes   31903 Aug 22 22:37 gemini-blueprint-io-
1.0.0.RELEASE.jar
  208 -rw-r--r-- 1 siepkes siepkes  208742 Oct  2 11:36 javax.jdo-3.2.0-m5.jar
 1336 -rw-r----- 1 siepkes siepkes 1363464 Oct 22 14:26
org.eclipse.osgi_3.7.1.R37x_v20110808-1106.jar
  320 -rw-r----- 1 siepkes siepkes  321428 Aug 18 16:50 org.springframework.aop-
3.0.6.RELEASE.jar
   56 -rw-r----- 1 siepkes siepkes   53082 Aug 18 16:50 org.springframework.asm-
3.0.6.RELEASE.jar
   36 -rw-r----- 1 siepkes siepkes   35557 Aug 18 16:50 org.springframework.aspects-
3.0.6.RELEASE.jar
  548 -rw-r----- 1 siepkes siepkes  556590 Aug 18 16:50 org.springframework.beans-
3.0.6.RELEASE.jar
  660 -rw-r----- 1 siepkes siepkes  670258 Aug 18 16:50 org.springframework.context-
3.0.6.RELEASE.jar
  104 -rw-r----- 1 siepkes siepkes  101450 Aug 18 16:50
org.springframework.context.support-3.0.6.RELEASE.jar
  380 -rw-r----- 1 siepkes siepkes  382184 Aug 18 16:50 org.springframework.core-
3.0.6.RELEASE.jar
  172 -rw-r----- 1 siepkes siepkes  169752 Aug 18 16:50 org.springframework.expression-
3.0.6.RELEASE.jar
  384 -rw-r----- 1 siepkes siepkes  386033 Aug 18 16:50 org.springframework.jdbc-
3.0.6.RELEASE.jar
  332 -rw-r----- 1 siepkes siepkes  334743 Aug 18 16:50 org.springframework.orm-
3.0.6.RELEASE.jar
    4 -rw-r----- 1 siepkes siepkes    1313 Aug 18 16:50 org.springframework.spring-
library-3.0.6.RELEASE.libd
  232 -rw-r----- 1 siepkes siepkes  231913 Aug 18 16:50
org.springframework.transaction-3.0.6.RELEASE.jar
```

Step 6 - Set up Eclipse

Here I will show how one can create a base for an application with our newly created target

platform.

Create a Target Platform in Eclipse by going to 'Window' → 'Preferences' → 'Plugin Development' → 'Target Platform' and press the 'Add' button. Select 'Nothing: Start with an empty target platform', give the platform a name and point it to the directory we put all the jars/bundles in. When you are done press the 'Finish' button. Indicate to Eclipse we want to use this new platform by ticking the checkbox in front of our newly created platform in the 'Target Platform' window of the 'Preferences' screen.

Create a new project in Eclipse by going to 'File' → 'New...' → 'Project' and Select 'Plug-in Project' under the 'Plugin development' leaf. Give the project a name (I'm going to call it 'nl.siepkes.test.project.a' in this example). In the radiobox options 'This plugin is targetted to run with:' select 'An OSGi framework' → 'standard'. Click 'Next'. Untick the 'Generate an activator, a Java class that....' and press 'Finish'.

Obviously Eclipse is not the mandatory IDE for the steps described above. Other technologies can be used instead. For this guide I used Eclipse because it is easy to explain, but for most of my projects I use Maven. If you have the Spring IDE plugin installed (which is advisable if you use Spring) you can add a Spring Nature to your project by right clicking your project and then clicking 'Spring Tools' → 'Add Spring Nature'. This will enable error detection in your Spring bean configuration file.

Create a directory called 'spring' in your 'META-INF' directory. In this directory create a Spring bean configuration file by right clicking the directory and click 'New...' → 'Other...'. A menu called 'New' will popup, select 'Spring Bean Configuration File'. Call the file beans.xml.

It is important to realize that the Datanucleus plugin system uses the Eclipse extensions system and NOT the plain OSGi facilities. There are two ways to make the DataNucleus plugin system work in a plain OSGi environment:

- Tell DataNucleus to use a simplified plugin manager which does not use the Eclipse plugin system (called "OSGiPluginRegistry").
- Add the Eclipse plugin system to the OSGi platform.

We are going to use the simplified plugin manager. The upside is that its easy to setup. The downside is that is less flexible then the Eclipse plugin system. The Eclipse plugin system allowes you to manage different version of DataNucleus plugins. With the simplified plugin manager you can have only *one* version of a DataNucleus plugin in your OSGi platform at any given time.

Declare a Persistence Manager Factory Bean inside the beans.xml:

```xml
<bean id="pmf"
class="nl.siepkes.util.DatanucleusOSGiLocalPersistenceManagerFactoryBean">
    <property name="jdoProperties">
        <props>
            <prop key="javax.jdo.PersistenceManagerFactoryClass"
>org.datanucleus.api.jdo.JDOPersistenceManagerFactory</prop>
            <!-- PostgreSQL DB connection settings. Add '?loglevel=2' to Connection
URL for JDBC Connection debugging. -->
            <prop key="javax.jdo.option.ConnectionURL"
>jdbc:postgresql://localhost/testdb</prop>
            <prop key="javax.jdo.option.ConnectionUserName">foo</prop>
            <prop key="javax.jdo.option.ConnectionPassword">bar</prop>

            <prop key="datanucleus.schema.autoCreateAll">true</prop>
            <prop key="datanucleus.schema.validateAll">true</prop>
            <prop key="datanucleus.rdbms.CheckExistTablesOrViews">true</prop>

            <prop key="datanucleus.plugin.pluginRegistryClassName"
>org.datanucleus.plugin.OSGiPluginRegistry</prop>
        </props>
    </property>
</bean>


<osgi:service ref="pmf" interface="javax.jdo.PersistenceManagerFactory" />
```

You can specify all the JDO/DataNucleus options you need following the above *prop*, *key* pattern. Notice the *osgi:service* line. This exports our persistence manager as an OSGi sevice and makes it possible for other bundles to access it. Also notice that the Persistence Manager Factory is not the normal *LocalPersistenceManagerFactoryBean* class, but instead the *OSGiLocalPersistenceManagerFactoryBean* class. The *OSGiLocalPersistenceManagerFactoryBean* is **NOT** part of the default DataNucleus distribution. So why do we need to use the *OSGiLocalPersistenceManagerFactoryBean* instead of the default *LocalPersistenceManagerFactoryBean* ? The default *LocalPersistenceManagerFactoryBean* is not aware of the OSGi environment and expects all classes to be loaded by one single classloader (this is the case in a normal Java environment without OSGi). This makes the *LocalPersistenceManagerFactoryBean* unable to locate its plugins. The *OSGiLocalPersistenceManagerFactoryBean* is a subclass of the *LocalPersistenceManagerFactoryBean* and is aware of the OSGi environment:

```java
public class OSGiLocalPersistenceManagerFactoryBean extends
LocalPersistenceManagerFactoryBean implements BundleContextAware {

    private BundleContext bundleContext;
    private DataSource dataSource;

    public DatanucleusOSGiLocalPersistenceManagerFactoryBean()
    {
    }
```

```java
    @Override
    protected PersistenceManagerFactory newPersistenceManagerFactory(String name)
    {
        return JDOHelper.getPersistenceManagerFactory(name, getClassLoader());
    }

    @Override
    protected PersistenceManagerFactory newPersistenceManagerFactory(Map props)
    {
        ClassLoader classLoader = getClassLoader();
        props.put("datanucleus.primaryClassLoader", classLoader);
        return JDOHelper.getPersistenceManagerFactory(props, classLoader);
    }

    private ClassLoader getClassLoader()
    {
        ClassLoader classloader = null;
        Bundle[] bundles = bundleContext.getBundles();
        for (int x = 0; x < bundles.length; x++)
        {
            if ("org.datanucleus.store.rdbms".equals(bundles[x].getSymbolicName()))
            {
                try
                {
                    classloader = bundles[x].loadClass
("org.datanucleus.ClassLoaderResolverImpl").getClassLoader();
                }
                catch (ClassNotFoundException e)
                {
                    e.printStackTrace();
                }
                break;
            }
        }
        return classloader;
    }

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

If we create an new, similar (Plug-in) project, for example 'nl.siepkes.test.project.b' we can import/use our Persistance Manager Factory service by specifying the following in its beans.xml:

```xml
<osgi:reference id="pmf" interface="javax.jdo.PersistenceManagerFactory" />
```

The Persistance Manager Factory (pmf) bean can then be injected into other beans as you normally would do when using Spring and JDO/DataNucleus together.

Step 7 - Accessing your services from another bundle

The reason why you are probably using OSGi is because you want to separate/modularize all kinds of code. A common use case is that you have your service layer in bundle A and another bundle, bundle B, who invokes methods in your service layer. Bundle B knows absolutely nothing about DataNucleus (ie. no imports and dependencies on DataNucleus or Datastore JDBC drivers) and will just call methods with signatures like 'public FooRecord getFooRecord(long fooId)'.

When you create such a setup and access a method in bundle A from bundle B you might be surprised to find out a ClassNotFound Exception is being thrown. The ClassNotFound exception will probably be about some DataNucleus or Datastore JDBC driver class not being found. How can bundle B complain about not finding implementation classes which only belong in bundle A (which has the correct imports) ? The reason for this is that when you invoke the method in bundle A from bundle B the classloader from bundle B is used to execute the method in bundle A. And since the classloader of bundle B does not have DataNucleus imports things go awry.

To solve this we need to change the ClassLoader in the ThreadContext which invokes the method in Bundle A. We could of course do this manually in every method in Bundle A but since we are already using Spring and AOP its much easier to do it that way. Create the following class (which is our aspect that is going to do the heavy lifting) in bundle A:

```java
package nl.siepkes.util;

/**
 * <p>
 * Aspect for setting the correct class loader when invoking a method in the
 * service layer.
 * </p>
 * <p>
 * When invoking a method from a bundle in the service layer of another bundle
 * the classloader of the invoking bundle is used. This poses the problem that
 * the invoking class loader needs to know about classes in the service layer of
 * the other bundle. This aspect sets the <tt>ContextClassLoader</tt> of the
 * invoking thread to that of the other bundle, the bundle that owns the method
 * in the service layer which is being invoked. After the invoke is completed
 * the aspect sets the <tt>ContextClassLoader</tt> back to the original
 * classloader of the invoker.
 * </p>
 *
 * @author Jasper Siepkes <jasper@siepkes.nl>
 *
 */
public class BundleClassLoaderAspect implements Ordered {

    private static final int ASPECT_PRECEDENCE = 0;
```

```java
    public Object setClassLoader(ProceedingJoinPoint pjp) throws Throwable {
        // Save a reference to the classloader of the caller
        ClassLoader oldLoader = Thread.currentThread().getContextClassLoader();
        // Get a reference to the classloader of the owning bundle
        ClassLoader serviceLoader = pjp.getTarget().getClass().getClassLoader();
        // Set the class loader of the current thread to the class loader of the
        // owner of the bundle
        Thread.currentThread().setContextClassLoader(serviceLoader);

        Object returnValue = null;

        try {
            // Make the actual call to the method.
            returnValue = pjp.proceed();
        } finally {
            // Reset the classloader of this Thread to the original
            // classloader of the method invoker.
            Thread.currentThread().setContextClassLoader(oldLoader);
        }

        return returnValue;
    }

    @Override
    public int getOrder() {
        return ASPECT_PRECEDENCE;
    }
}
```

Add the following to you Spring configuration in bundle A:

```xml
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
    <tx:method name="get*" read-only="true" />
    <tx:method name="*" />
    </tx:attributes>
</tx:advice>

<aop:pointcut id="fooServices" expression="execution(* nl.siepkes.service.*.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServices" />

    <!-- Ensures the class loader of this bundle is used to invoke public methods in
the service layer of this bundle. -->
    <aop:aspect id="bundleLoaderAspect" ref="bundleLoaderAspectBean">
    <aop:around pointcut-ref="fooServices" method="setClassLoader"/>
    </aop:aspect>
</aop:config>
```

Now all methods in classes in the package 'nl.siepkes.service' will always use the class loader of

bundle A.

Using DataNucleus with Eclipse RCP

This guide was written by Stuart Robertson.

Using DataNucleus inside an Eclipse plugin (that is, Eclipse's Equinox OSGi runtime) should be simple, because DataNucleus is implemented as a collection of OSGi bundles. My early efforts to use DataNucleus from within my Eclipse plugins all ran into problems. First classloader problems of various kinds began to show themselves. This was revealed in a post on the old DataNucleus forum (now closed). My initial faulty configuration was as follows:

```
model
  src/main/java/...*.java     (persistent POJO classes, enhanced using Maven
DataNucleus plugin)
  src/main/resources/datanucleus.properties* (PMF properties)

rcp.jars
  plugin.xml
  META-INF/
    MANIFEST.MF   (OSGi bundle manifest)
  lib/
    datanucleus-core-XXX.jar
    ...
    spring-2.5.jar

rcp.ui
  plugin.xml
  META-INF/
    MANIFEST.MF   (OSGi bundle manifest)
```

Using the standard pattern, I had created a "jars" plugin whose only purpose in life was to provide a way to bring all of the 3rd party jars that my "model" depends on into the Eclipse plugin world. Each of the jars in the "jars" project's lib directory were also added to the MANIFEST.MF "Bundle-ClassPath" section as follows:

```
Bundle-ClassPath:* lib\asm-3.0.jar,
lib\aspectjtools-1.5.3.jar,
lib\commons-dbcp-1.2.2.jar,
lib\commons-logging-1.1.1.jar,
lib\commons-pool-1.3.jar,
lib\geronimo-spec-jta-1.0.1B-rc2.jar,
lib\h2-1.0.63.jar,
lib\jdo2-api-2.1-SNAPSHOT.jar,
lib\datanucleus-core-XXX.jar,
lib\datanucleus-rdbms-XXX.jar,
lib\...*
lib\log4j-1.2.14.jar,
lib\model-1.0.0-SNAPSHOT.jar,
lib\javax.persistence-2.1.jar,
lib\spring-2.5.jar
```

Notice that the *rcp.jars* plugin's lib directory contains **model-1.0.0-SNAPSHOT.jar** - this is the jar containing my enhanced persistent classes and PMF properties file (which I called *datanucleus.properties*). Also, *all* of the packages from *all* of the jars listed in the Bundle-Classpath were exported using the Export-Package bundle-header.

Note, that the plugin.xml file in the "jars" project is an empty plugin.xml file containing only <plugin></plugin>, used only to trick Eclipse into using the Plugin Editor to open the `MANIFEST.MF` file so the bundle info can be edited in style.

The *rcp.ui plugin* depends on the *rcp.jars* so that it can "see" all of the necessary classes. Inside the Bundle Activator class in my UI plugin I initialized DataNucleus as normal, creating a PersistenceManagerFactory from the embedded datanucleus.properties file.

It all looks really promising, but doesn't work due to all kinds of classloading issues.

DataNucleus jars as plugins

The first part of the solution was to use the DataNucleus as a set of Eclipse plugins. Initially I wasn't sure where to get MANIFEST.MF and plugin.xml files to do this, but I later discovered that each of the datanucleus jar files are already packaged as Eclipse plugins. Open any of the datanucleus jar files up and you'll see an OSGi manifest and Eclipse plugin.xml. All that was needed was to copy `datanucleus-XXX.jar` into `$ECLIPSE_HOME/plugins` directory and restart Eclipse.

Once this was done, I removed the datanucleus jar files from my lib/ directory and instead modified my jars plugin, removing the datanucleus jars and all datanucleus packages from Bundle-Classpath and Export-Package. Next, I modified my *rcp.ui plugin* to depend not only on *rcp.jars*, but also on all of the **datanucleus** plugins. The relevant section of my rcp.ui plugin's manifest were changed to:

```
Require-Bundle: org.eclipse.core.runtime,
org.datanucleus,
org.datanucleus.enhancer,
org.datanucleus.store.rdbms,
```

This moved things along, resulting in the following message:

```
javax.jdo.JDOException: Class org.datanucleus.store.rdbms.RDBMSManager was not found
in the CLASSPATH. Please check your specification and your CLASSPATH.
```

Turns out that the class that could not be found was not `org.datanucleus.store.rdbms.RDBMSManager`, but rather my H2 database driver class. I figured the solution might lie in using Eclipse's buddy-loading mechanism to allow the **org.datanucleus.store.rdbms** plugin to see my JDBC driver, which is was packaged into my 'jars' plugin. Thus, I added the following to *rcp.ui*'s MANIFEST.MF:

```
Eclipse-RegisterBuddy: org.datanucleus.store.rdbms
```

That too, didn't work. Checking the org.datanucleus.store.rdbms MANIFEST.MF showed no 'Eclipse-BuddyPolicy: registered' entry, so *Eclipse-RegisterBuddy: org.datanucleus.store.rdbms* wouldn't have helped anyway. If you are new to Eclipse's classloading ways, I can highly recommend you read A Tale of Two VMs, as you'll likely run into the need for buddy-loading sooner or later.

PrimaryClassLoader saves the day

Returning to Erik Bengtson's DataNucleus forum example gave me inspiration:

```
//set classloader for driver (using classloader from the "rcp.jars" bundle)
ClassLoader clrDriver = Platform.getBundle("rcp.jars").loadClass("org.h2.Driver"
).getClassLoader();
map.put("org.datanucleus.primaryClassLoader", clrDriver);

//set classloader for DataNucleus (using classloader from the "org.datanucleus"
bundle)
ClassLoader clrDN = Platform.getBundle("org.datanucleus").loadClass
("org.datanucleus.api.jdo.JDOPersistenceManagerFactory").getClassLoader()

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(map, clrDN);
```

With the above change made, things worked. So, in summary

- Don't embed DataNucleus jars inside your plugin

- Do install DataNucleus jars into Eclipse/plugins and add dependencies to them from your plugin's MANIFEST

- Do tell DataNucleus which classloader to use for both its primaryClassLoader and for its own implementation

DataNucleus + Eclipse RCP + Spring

This guide was written by Stuart Robertson.

In my application, I have used [Spring]'s elegant JdoDaoSupport class to implement my DAOs, have used Spring's BeanFactory to instantiate PersistenceManagerFactory and DAO instances and have set up declarative transaction management. See the [Spring documentation section 12.3] if you are unfamiliar with Spring's JDO support. I assumed, naively, that since my code all worked when built and unit-tested in a plain Java world (with Maven 2 building my jars and running my unit-tests), that it would work inside Eclipse. I found out above that using DataNucleus inside Eclipse RCP application needs a little special attention to classloading. Once this has been taken care of, you'll know that you need to provide your PersistenceManagerFactory with the correct classloader to use as "primaryClassLoader". However, since everything is going to be instantiated by the Spring bean container, it somehow has to know what "the correct classloader" is. The recipe is fairly simple.

Add a Factory-bean and factory-method

At first I wasn't sure what needed doing, but a little browsing of the Spring documentation revealed what I needed (see [section 3.2.3.2.3. Instantiation using an instance factory method]). Spring provides a mechanism whereby a Spring beans definition file (beans.xml, in my case) can defer the creation of an object to either a static method on some factory class, or a non-static (instance) method one some factory bean. The following quote from the Spring documentation describes how things are meant to work:

In a fashion similar to instantiation via a static factory method, instantiation using an instance factory method is where a non-static method of an existing bean from the container is invoked to create a new bean. To use this mechanism, the 'class' attribute must be left empty, and the 'factory-bean' attribute must specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. The name of the factory method itself must be set using the 'factory-method' attribute.

The example bean definitions below show how a bean can be created using this pattern:

```
<!-- the factory bean, which contains a method called createService() -->
<bean id="serviceLocator" class="com.foo.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="exampleBean" factory-bean="serviceLocator" factory-method="createService"/>
```

Add a little ClassLoaderFactory

In my case, I replaced the "serviceLocator" factory bean with a "classloaderFactory" bean with factory-methods that return Classloader instances, as shown below:

```java
/**
 * Used as a bean inside the Spring config so that the correct classloader can be
"wired" into the PersistenceManagerFactory bean.
 */
public class ClassLoaderFactory
{
    /** Used in beans.xml to set the PMF's primaryClassLoaderResolver property. */
    public ClassLoader jdbcClassloader()
    {
        return getClassloaderFromClass("org.h2.Driver");
    }

    public ClassLoader dnClassloader()
    {
        return getClassloaderFromClass
("org.datanucleus.api.jdo.JDOPersistenceManagerFactory");
    }

    private ClassLoader getClassloaderFromClass(String className)
    {
        try
        {
            ClassLoader classLoader = Activator.class.getClassLoader().loadClass
(className).getClassLoader();
            return classLoader;
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
            throw new RuntimeException(e.getMessage(), e);
        }
    }
}
```

The two public methods, jdbcClassloader() and dnClassloader(), ask the bundle Activator to load a particular class, and then return the Classloader that was used to load the class. Note that Activator is the standard bundle activator created by Eclipse. OSGi classloading is based on a setup where each bundle has its own classloader. For example, if bundle A depends on bundles B and C, attempting to load a class (ClassC, say) provided by bundle C will result in bundle A's classloader delegating the class-load to bundle C. Calling getClassLoader() on the loaded ClassC will return bundle C's classloader, not bundle A's classloader. And this is exactly the behaviour we need. Thus, asking Activator's classloader to load "org.h2.Driver" will ultimately delegate the loading to the classloader associated with the bundle that contains the JDBC driver classes. Likewise with "org.datanucleus.api.jdo.JDOPersistenceManagerFactory".

Mix well

Now we have all of the pieces needed to configure our Spring beans. The bean definitions below are a part of a larger beans.xml file, but show the relevant setup. The list below describes each of the

beans working from top to bottom, where the text in bold is the bean id:

- **placeholderConfigurer** : This is a standard Spring property configuration mechanism that loads a properties file from the classpath location "classpath:/config/jdbc.${datanucleus.profile}.properties", where ${datanucleus.profile} represents the value of the "datanucleus.profile" environment variable which I set externally so that I can switch between in-memory, on-disk embedded or on-disk server DB configurations.

- **dataSource** : A JDBC DataSource (using Apache DBCP's connection pooling DataSource). Values for the properties ${jdbc.driverClassName}, ${jdbc.url}, etc are obtained from the properties file that was loaded by **placeholderConfigurer**.

- **pmf** : The DataNucleus PersistenceManagerFactory (implementation) that underpins the entire persistence layer. It's a fairly standard setup, with a reference to **dataSource** being stored in connectionFactory. The important part for this discussion is the *primaryClassLoaderResolver* part, which stores a reference to a Classloader instance (a Classloader "bean", that is).

- **classloaderFactory** and **jdbcClassloader** : Here we pull in the factory-bean pattern discussed above. When asked for the **jdbcClassloader** bean (which is a Classloader instance), Spring will defer to **classloaderFactory**, creating an instance of ClassLoaderFactory and then calling its jdbcClassloader() method to obtain the Classloader that is to become the **jdbcClassloader** bean. This works, because the the Spring jar is able to "see" my ClassLoaderFactory class. If the Spring jar is contained in one bundle, A, say, and your factory class is in some other bundle, B, say, then you may encounter ClassNotFoundException if bundle A doesn't depend on bundle B. This is normally the case if you follow the "jars plugin" pattern, creating a single plugin to house all third-party jars. In this case, you will need to add "Eclipse-BuddyPolicy: registered" to the "jars" plugin's manifest, and then add "Eclipse-RegisterBuddy: <jars.bundle.symbolicname>" to the manifest of the bundle that houses your factory class (where <jars.bundle.symbolicname> must be replaced with the actual symbolic name of the bundle). See A Tale of Two VMs if this is Greek to you.

```
<!-- ====== JDO PERSISTENCE INFRASTRUCTURE ====== -->
<bean id="placeholderConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
    p:location="classpath:/config/jdbc.${datanucleus.profile}.properties" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}" />

<bean id="pmf" class="org.datanucleus.api.jdo.JDOPersistenceManagerFactory"
    destroy-method="close"
    p:connectionFactory-ref="dataSource"
    p:attachSameDatastore="true"
    p:autoCreateColumns="true"
    p:autoCreateSchema="true"
    p:autoStartMechanism="None"
    p:detachAllOnCommit="true"
    p:detachOnClose="false"
    p:nontransactionalRead="true"
    p:stringDefaultLength="255"
    p:primaryClassLoaderResolver-ref="jdbcClassloader" />

<bean id="classloaderFactory" class="rcp.model.ClassLoaderFactory" />

<!-- the bean to be created via the factory bean -->
<bean id="jdbcClassloader"
    factory-bean="classloaderFactory"
    factory-method="jdbcClassloader" />
```

Enjoy

Now that the hard-work is done, we can ask Spring to do its magic:

```
private void loadSpringBeans()
{
    if (beanFactory == null)
    {
        beanFactory = new ClassPathXmlApplicationContext("/config/beans.xml",
Activator.class);
    }
    this.daoFactory = (IDAOFactory) beanFactory.getBean("daoFactory");
}

private void testDAO()
{

    IAccountDAO accountsDAO = this.daoFactory.accounts();
    accountsDAO.persist(entities.newAccount("Account A", AccountType.Asset));
    accountsDAO.persist(entities.newAccount("Account B", AccountType.Bank));
    List<IAccount> accounts = accountsDAO.findAll();
}
```

Finally, I should clarify things by mentioning that in my code, my bundle Activator provides the loadSpringBeans() method and calls it when the bundle is started. Other classes, such as the main application, then use Activator.getDefault().getDAOFactory() to obtain a reference to IDAOFactory, which is another Spring bean that provides a central point of reference to all of the DAOs in the system. All of the DAOs themselves are Spring beans too.

Postscript

Someone asked to see the complete applicationContext.xml (referred to as /config/beans.xml in the loadSpringBeans() method above), so here it is:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.1.xsd
    http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">


    <!-- Enable the use of @Autowired annotations. -->
```

```xml
<context:annotation-config />

<!-- ====== MAIN ENTRY-POINTS ====== -->
<bean
    id="daoFactory"
    class="ca.eulogica.bb.model.dao.impl.DAOFactory"
    p:accountDAO-ref="accountDAO"
    p:budgetDAO-ref="budgetDAO"
    p:budgetItemDAO-ref="budgetItemDAO"
    p:commodityDAO-ref="commodityDAO"
    p:institutionDAO-ref="institutionDAO"
    p:splitDAO-ref="splitDAO"
    p:transactionDAO-ref="transactionDAO" />

<bean
    id="entityFactory"
    class="ca.eulogica.bb.model.entities.impl.EntityFactory" />

<bean
    id="servicesFactory"
    class="ca.eulogica.bb.model.services.impl.ServicesFactory"
    p:accountService-ref="accountService"
    p:transactionService-ref="transactionService" />

<!-- ====== BUSINESS SERVICES ====== -->
<bean
    id="accountService"
    class="ca.eulogica.bb.model.services.impl.AccountService"
    p:DAOFactory-ref="daoFactory"
    p:entityFactory-ref="entityFactory" />

<bean
    id="transactionService"
    class="ca.eulogica.bb.model.services.impl.TransactionService"
    p:DAOFactory-ref="daoFactory"
    p:entityFactory-ref="entityFactory" />

<!-- ====== DAO ====== -->
<bean
    id="accountDAO"
    class="ca.eulogica.bb.model.dao.impl.AccountDAO"
    p:persistenceManagerFactory-ref="pmf" />

<bean
    id="budgetDAO"
    class="ca.eulogica.bb.model.dao.impl.BudgetDAO"
    p:persistenceManagerFactory-ref="pmf" />

<bean
    id="budgetItemDAO"
    class="ca.eulogica.bb.model.dao.impl.BudgetItemDAO"
```

```xml
                    p:persistenceManagerFactory-ref="pmf" />

    <bean
        id="commodityDAO"
        class="ca.eulogica.bb.model.dao.impl.CommodityDAO"
        p:persistenceManagerFactory-ref="pmf" />

    <bean
        id="institutionDAO"
        class="ca.eulogica.bb.model.dao.impl.InstitutionDAO"
        p:persistenceManagerFactory-ref="pmf" />

    <bean
        id="splitDAO"
        class="ca.eulogica.bb.model.dao.impl.SplitDAO"
        p:persistenceManagerFactory-ref="pmf" />

    <bean
        id="transactionDAO"
        class="ca.eulogica.bb.model.dao.impl.TransactionDAO"
        p:persistenceManagerFactory-ref="pmf" />

    <!-- ====== TRANSACTION MANAGEMENT ====== -->
    <bean
        id="txManager"
        class="org.springframework.orm.jdo.JdoTransactionManager"
        p:persistenceManagerFactory-ref="pmf" />

    <tx:advice
        id="txAdvice"
        transaction-manager="txManager">
        <tx:attributes>
            <tx:method
                name="get*"
                propagation="REQUIRED"
                read-only="true" />
            <tx:method
                name="*"
                propagation="REQUIRED" />
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <aop:pointcut
            id="daoMethodsPointcut"
            expression="execution(* ca.eulogica.bb.model.dao.impl.*.*(..))" />
        <aop:advisor
            id="daoMethodsAdvisor"
            advice-ref="txAdvice"
            pointcut-ref="daoMethodsPointcut" />
    </aop:config>
```

```xml
    <aop:config>
        <aop:pointcut
            id="serviceMethodsPointcut"
            expression="execution(* ca.eulogica.bb.model.services.*.*(..))" />
        <aop:advisor
            id="serviceMethodsAdvisor"
            advice-ref="txAdvice"
            pointcut-ref="serviceMethodsPointcut" />
    </aop:config>

    <!-- ====== JDO PERSISTENCE INFRASTRUCTURE ====== -->
    <bean id="placeholderConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
        p:location="classpath:/config/jdbc.${datanucleus.profile}.properties" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}" />

    <bean id="pmf" class="org.datanucleus.api.jdo.JDOPersistenceManagerFactory"
        destroy-method="close"
        p:connectionFactory-ref="dataSource"
        p:attachSameDatastore="true"
        p:autoCreateColumns="true"
        p:autoCreateSchema="true"
        p:autoStartMechanism="None"
        p:detachAllOnCommit="true"
        p:detachOnClose="false"
        p:nontransactionalRead="true"
        p:stringDefaultLength="255"
        p:primaryClassLoaderResolver-ref="jdbcClassloader" />

    <bean id="classloaderFactory" class="budgetbuddy.rcp.model.ClassLoaderFactory" />

    <!-- the bean to be created via the factory bean -->
    <bean id="jdbcClassloader"
        factory-bean="classloaderFactory"
        factory-method="jdbcClassloader" />

</beans>
```

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

JPA 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
```

PersistenceManagerFactory usage

Creation of PersistenceManagerFactory 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 setting the property **datanucleus.autoStartMechanism** to *None*. This means that it won't try to load up the classes (or better said the metadata of the classes) handled the previous time that this schema was used. If this isn't an issue for your application then you can make this change. Please refer to the Auto-Start Mechanism for full details.

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 PMF as a whole (using the persistence properties **javax.jdo.mapping.Catalog**, **javax.jdo.mapping.Schema**) 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.

PersistenceManager usage

Clearly the structure of your application will have a major influence on how you utilise a PersistenceManager. 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 PM's is not an expensive process and use of multiple threads writing to the same persistence manager should be avoided.

Make sure that you always close the PersistenceManager 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 PersistenceManager if you have the persistence property **datanucleus.detachOnClose** set to *true* 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.

  - DataNucleus verifies if newly persisted objects are memory reachable on commit, if they are not, they are removed from the database. This process mirrors the garbage collection, where objects not referenced are garbage collected or removed from memory. Reachability is expensive because it traverses the whole object tree and may require reloading data from database. If reachability is not needed by your application, you should disable it. To disable reachability set the persistence property **datanucleus.persistenceByReachabilityAtCommit** to *false*.

  - DataNucleus will, by default, perform a check on any bidirectional relations to make sure that they are set at both sides at commit. If they aren't set at both sides then they will be made consistent. This check process can involve the (re-)loading of some instances. You can skip this step if you always set *both sides of a relation* by setting the persistence property **datanucleus.manageRelationships** to *false*.

  - 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**.

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 **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 **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 *increment* strategy should be used instead. Better still would be to choose *native* and let DataNucleus decide for you.

The **NATIVE** identity generator value is the recommended choice since this will allow DataNucleus to decide which value 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 PMF** - 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 PMF** - 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 PMF 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 PersistenceManager. The second time an application requests the same objects from the PersistenceManager 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 PersistenceManager

Reading fields of unmanaged objects (outside the scope of a *PersistenceManager*) 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 *makeTransient* to get *transient* versions of the objects. Note that to recurse you need to call the *makeTransient* method which has a boolean argument "useFetchPlan".

```
Object pc = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
    pm.makeTransient(pc);

    pm.currentTransaction().commit();
}
finally
{
    pm.close();
}
//read the persistent object here
System.out.prinln(pc.getName());
```

- With persistence property **datanucleus.RetainValues** set to *true*.

```
Object pc = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().setRetainValues(true);
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);

    pm.currentTransaction().commit();
}
finally
{
    pm.close();
}
//read the persistent object here
System.out.prinln(pc.getName());
```

- Use *detachCopy* method to return detached instances.

```
Object copy = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    Object pc = pm.getObjectById(id);
    copy = pm.detachCopy(pc);

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

- Use *detachAllOnCommit.*

```
Object pc = null;
try
{
    PersistenceManager pm = pmf.getPersistenceManager();
    pm.setDetachAllOnCommit(true);
    pm.currentTransaction().begin();

    //retrieve in some way the object, query, getObjectById, etc
    pc = pm.getObjectById(id);
    pm.currentTransaction().commit(); // Object "pc" is now detached
}
finally
{
    pm.close();
}
//read or change the detached object here
System.out.prinln(pc.getName());
```

The most expensive in terms of performance is the *detachCopy* because it makes copies of persistent objects. The advantage of detachment (via *detachCopy* or *detachAllOnCommit*) is that changes made outside the transaction can be further used to update the database in a new transaction. The other methods also allow changes outside of the transaction, but the changed instances can't be used to update the database.

With *RetainValues=true* and *makeTransient* no object copies are made and the object values are set down in instances when the PersistenceManager disassociates them. Both methods are equivalent in performance, however the *makeTransient* method will set the values of the object during the

instant the *makeTransient* method is invoked, and the *RetainValues=true* will set values of the object during commit.

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

Queries usage

Make sure you close all query results after you have finished with them. Failure to do so will result in significant memory leaks in your application.

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 JDO the default "maximum fetch depth" is 1.

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 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

Extension

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 JDO 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

> 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 PMF1 to PMF2. These PMFs are created in the normal way so, as mentioned above, PMF1 could be for a MySQL datastore, and PMF2 for XML. By default this will replicate the complete object graphs reachable from these specified types.

```
import org.datanucleus.api.jdo.JDOReplicationManager;

...

JDOReplicationManager replicator = new JDOReplicationManager(pmf1, pmf2);
replicator.replicate(new Class[]{Product.class, Employee.class});
```

Example without using the JDOReplicationManager helper

If we just wanted to use pure JDO, we would handle replication like this. Let's take an example

```java
public class ElementHolder
{
    long id;
    private Set elements = new HashSet();

    ...
}

public class Element
{
    String name;

    ...
}

public class SubElement extends Element
{
    double value;

    ...
}
```

so we have a 1-N unidirectional (Set) relation, and we define the metadata like this

```xml
<jdo>
    <package name="mydomain.samples">
        <class name="ElementHolder" identity-type="application" detachable="true">
            <inheritance strategy="new-table"/>
            <field name="id" primary-key="true"/>
            <field name="elements" persistence-modifier="persistent">
                <collection element-type="mydomain.samples.Element"/>
                <join/>
            </field>
        </class>

        <class name="Element" identity-type="application" detachable="true">
            <inheritance strategy="new-table"/>
            <field name="name" primary-key="true"/>
        </class>

        <class name="SubElement">
            <inheritance strategy="new-table"/>
            <field name="value"/>
        </class>
    </package>
</jdo>
```

so in our application we create some objects in *datastore1*, like this

```
PersistenceManagerFactory pmf1 = JDOHelper.getPersistenceManagerFactory
("dn.1.properties");
PersistenceManager pm1 = pmf1.getPersistenceManager();
Transaction tx1 = pm1.currentTransaction();
Object holderId = null;
try
{
    tx1.begin();

    ElementHolder holder = new ElementHolder(101);
    holder.addElement(new Element("First Element"));
    holder.addElement(new Element("Second Element"));
    holder.addElement(new SubElement("First Inherited Element"));
    holder.addElement(new SubElement("Second Inherited Element"));
    pm1.makePersistent(holder);

    tx1.commit();
    holderId = JDOHelper.getObjectId(holder);
}
finally
{
    if (tx1.isActive())
    {
        tx1.rollback();
    }
    pm1.close();
}
```

and now we want to replicate these objects into *datastore2*, so we detach them from *datastore1* and attach them to *datastore2*, like this

```java
// Detach the objects from "datastore1"
ElementHolder detachedHolder = null;
pm1 = pmf1.getPersistenceManager();
tx1 = pm1.currentTransaction();
try
{
    pm1.getFetchPlan().setGroups(new String[] {FetchPlan.DEFAULT, FetchPlan.ALL});
    pm1.getFetchPlan().setMaxFetchDepth(-1);

    tx1.begin();

    ElementHolder holder = (ElementHolder) pm1.getObjectById(holderID);
    detachedHolder = (ElementHolder) pm1.detachCopy(holder);

    tx1.commit();
}
finally
{
    if (tx1.isActive())
    {
        tx1.rollback();
    }
    pm1.close();
}

// Attach the objects to datastore2
PersistenceManagerFactory pmf2 = JDOHelper.getPersistenceManagerFactory
("dn.2.properties");
PersistenceManager pm2 = pmf2.getPersistenceManager();
Transaction tx2 = pm2.currentTransaction();
try
{
    tx2.begin();

    pm2.makePersistent(detachedHolder);

    tx2.commit();
}
finally
{
    if (tx2.isActive())
    {
        tx2.rollback();
    }
    pm2.close();
}
```

That's all there is. These objects are now replicated into *datastore2*. Clearly you can extend this basic idea and replicate large amounts of data.

Data Federation

Extension

By default JDO provides a [PersistenceManagerFactory](#) (PMF) to represent a single datastore. Some applications need access to multiple datastores and the standard way of handling this is to have one PMF for each datastore.

As an alternative DataNucleus allows having a PMF represent multiple datastores.

⚠️ This is functionality that is work-in-progress and only tested for basic persist/retrieve operations using different schemas of the same datastore. It may work for some things but you need to treat it with caution.

ℹ️ Obviously if you have relations between one object in one datastore and another object in another datastore you cannot have *foreign-keys* (or equivalent).

Defining Primary and Secondary Datastores

You could specify the datastores to be used for the PMF like this. Here we have `datanucleus.properties` defining the **primary datastore**

```
javax.jdo.option.ConnectionURL=jdbc:mysql://127.0.0.1/nucleus?useServerPrepStmts=false
javax.jdo.option.ConnectionUserName=mysql
javax.jdo.option.ConnectionPassword=

datanucleus.datastore.store2=datanucleus2.properties
```

You note that this refers to a **store2**, which is defined by `datanucleus2.properties`. So the **secondary datastore** is then defined by another file.

```
javax.jdo.option.ConnectionURL=mongodb:/nucleus
```

Defining which class is persisted to which datastore

Now we need to notate which class is persisted to the **primary** datastore and which is persisted to **secondary** datastores. We do it like this, for the classes persisted to the secondary datastore.

```
@PersistenceCapable
@Extension(vendorName="datanucleus", key="datastore", value="store2")
public class MyOtherClass
{
    ...
}
```

So for any persistence of objects of type *MyOtherClass*, they will be persisted into the MongoDB secondary datastore.

Java Security

The Java Security Manager can be used with DataNucleus JDO to provide a security platform to sensitive applications.

To use the Security Manager, specify the *java.security.manager* and *java.security.policy* arguments when starting the JVM. e.g.

```
java -Djava.security.manager
-Djava.security.policy==/etc/apps/security/security.policy ...
```

Note that when you use *-Djava.security.policy==...* (double equals sign) you override the default JVM security policy files, while if you use *-Djava.security.policy=...* (single equals sign), you append the security policy file to any existing ones.

The following is a sample security policy file to be used with DataNucleus.

```
grant codeBase "file:${/}javax.jdo-3.2*.jar" {

    //jdo API needs datetime (timezone class needs the following)
    permission java.util.PropertyPermission "user.country", "read";
    permission java.util.PropertyPermission "user.variant", "read";
    permission java.util.PropertyPermission "user.timezone", "read,write";
    permission java.util.PropertyPermission "java.home", "read";
};
grant codeBase "file:${/}datanucleus*.jar" {

     //jdo
    permission javax.jdo.spi.JDOPermission "getMetadata";
    permission javax.jdo.spi.JDOPermission "setStateManager";

    //DataNucleus needs to get classloader of classes
    permission java.lang.RuntimePermission "getClassLoader";

    //DataNucleus needs to detect the java and os version
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "os.name", "read";

    //DataNucleus reads these system properties
    permission java.util.PropertyPermission "datanucleus.*", "read";
    permission java.util.PropertyPermission "javax.jdo.*", "read";

    //DataNucleus runtime enhancement (needs read access to all jars/classes in
classpath,
    // so use <<ALL FILES>> to facilitate config)
    permission java.lang.RuntimePermission "createClassLoader";
    permission java.io.FilePermission "<<ALL FILES>>", "read";

    //DataNucleus needs to read manifest files (read permission to location of
MANIFEST.MF files)
    permission java.io.FilePermission "${user.dir}${/}-", "read";
    permission java.io.FilePermission "<<ALL FILES>>", "read";

    //DataNucleus uses reflection!!!
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
    permission java.lang.RuntimePermission "accessDeclaredMembers";
};

grant codeBase "file:${/}datanucleus-hbase*.jar" {

    //HBASE does not run in a doPrivileged, so we do...
    permission java.net.SocketPermission "*", "connect,resolve";
};
```

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 PMF and one for each PM. You access these as follows

```
JDOPersistenceManagerFactory dnPMF = (JDOPersistenceManagerFactory)pmf;
FactoryStatistics stats = dnPMF.getNucleusContext().getStatistics();
... (access the statistics information)


JDOPersistenceManager dnPM = (JDOPersistenceManager)pm;
ManagerStatistics stats = dnPM.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 the JRE since v1.5, and you can easily activate DataNucleus MBeans registration by creating your PMF with the persistence property **datanucleus.jmxType** as *default*

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 JDO PMF instantiation). To see the full list of DataNucleus MBeans, refer to the javadocs.

mx4j

To enable management using MX4J you must specify the persistence property **datanucleus.jmxType** as *mx4j* when creating the PMF, and have the `mx4j` and `mx4j-tools` jars in the CLASSPATH.

DataNucleus Logging

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** - if you have Log4J in the CLASSPATH, [Apache Log4J](#) will be used

- **java.util.logging** - 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](#) or [Gitter](#)

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

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.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).