JDO Getting Started Guide (v5.0)

Table of Contents

Developing applications is, in general, a complicated task, involving many components. Developing all of these components can be very time consuming. The Java Data Objects API (JDO) was designed to alleviate some of this time spent, providing an API to allow java developers to persist object-oriented data into any database, and providing a query language using the same Java syntax as the developer is already familiar with.

DataNucleus JDO provides an implementation of this JDO *standard*, allowing you, the user, to persist your object-oriented data to not only the RDBMS datastores the standard was intended for, but also to a wide range of other datastores. These include popular map stores such as Cassandra and HBase, the Neo4j graph store, spreadsheets in Excel or OpenDocument formats, JSON formatted Amazon and Google Storage options, the popular MongoDB JSON-like document store, as well as ubiquitous LDAP and more besides.

DataNucleus doesn't purport to be the best solution to every problem. For example, where you want to bulk persist large amounts of data then other solutions that get closer to the datastore API would be more appropriate. Where you want to tailor the precise query sent to the datastore to take advantage of some datastore-specific feature is another situation in which you may find a hand-crafted solution more appropriate. That said, the range of capabilities of DataNucleus JDO cover a wide range of use-cases, the barrier to entry for use of DataNucleus us very low. You do not need to necessarily be an expert in all features of the chosen datastore to use it. It shields you from the majority of the more routine handling, whilst still letting you have a high degree of control over its behaviour and we hope that you benefit from its features.

Key Points

There are some key points to bear in mind when starting using JDO for java persistence.

- Your classes should be exactly that, *your* classes. DataNucleus imposes little to nothing on you. The DataNucleus enhancer provides for adding on a *default constructor* if you haven't provided one.
- Your JDO persistent classes need bytecode enhancing for use in the persistence process, but this can be an automatic post-compilation step.
- To persist objects of classes you firstly need to **define which classes are persistable, and how they are persisted**. Start under the JDO Mapping Guide
- Use of JDO requires a PersistenceManagerFactory to access the datastore.
- The persistence itself is controlled by a PersistenceManager and each object to be persisted will have different lifecycle states that you need to have an understanding of.
- You retrieve objects either by their identity, or using a query. With JDO you can use JDOQL or SQL query languages
- You will need `javax.jdo` as well as `datanucleus-api-jdo`, `datanucleus-core` and the `datanucleus-XXX` jar for whichever datastore you are using.

Understanding the JARs

DataNucleus has a modular architecture and you will need to make use of multiple JARs in your application, as follows

- `javax.jdo.jar` : This is the JDO API. This is basically a collection of interfaces, annotations and helper classes.
- `datanucleus-api-jdo.jar` : This is DataNucleus' implementation of the JDO API. It implements the interfaces defined in *javax.jdo.jar*.
- `datanucleus-core.jar` : This provides the basic DataNucleus persistence mechanism, and is required by all DataNucleus plugins.
- `datanucleus-{datastore}.jar` ({datastore} is 'rdbms', 'mongodb', 'cassandra', etc) : This provides persistence to the specific type of datastore that the JAR is for.
- `datanucleus-jdo-query.jar` : This provides an *annotation processor* and is used by the JDO Typed Query mechanism to generate the Q classes used at runtime.

There are various additional JARs that can be used, providing support for additional (non-standard) types, or features (such as third-party caching products).

JDO Tutorial (v5.0)

Download Source Code (GitHub)

Background

An application can be JDO-enabled via many routes depending on the development process of the project in question. For example the project could use Eclipse as the IDE for developing classes. In that case the project would typically use the DataNucleus Eclipse plugin. Alternatively the project could use Ant, Maven or some other build tool. In this case this tutorial should be used as a guiding way for using DataNucleus in the application. The JDO process is quite straightforward.

- Step 0 : Download DataNucleus AccessPlatform
- Step 1 : Define their persistence definition using Meta-Data.
- Step 2 : Define the "persistence-unit"
- Step 3 : Compile your classes, and instrument them (using the DataNucleus enhancer).
- Step 4 : Write your code to persist your objects within the DAO layer.
- Step 5 : Run your application.

We will take this further with 2 optional steps, showing how you can control the generated schema, and indeed how you generate the schema for your classes

- Step 6 : Controlling the schema
- Step 7 : Generate the database tables where your classes are to be persisted

The tutorial guides you through this. You can obtain the code referenced in this tutorial from SourceForge (one of the files entitled "datanucleus-samples-jdo-tutorial-*").

Step 0 : Download DataNucleus AccessPlatform

You can download DataNucleus in many ways, but the simplest is to download the distribution zip appropriate to your datastore. You can do this from SourceForge DataNucleus download page. When you open the zip you will find DataNucleus jars in the *lib* directory, and dependency jars in the *deps* directory.

Step 1 : Take your model classes and mark which are persistable

For our tutorial, say we have the following classes representing a store of products for sale.

```
package org.datanucleus.samples.jdo.tutorial;

public class Inventory
{
    String name = null;
    Set<Product> products = new HashSet<>();

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

    public Set<Product> getProducts() {return products;}
}
```

```
package org.datanucleus.samples.jdo.tutorial;

public class Product
{
    long id;
    String name = null;
    String description = null;
    double price = 0.0;

    public Product(String name, String desc, double price)
    {
        this.name = name;
        this.description = desc;
        this.price = price;
    }
}
```

```
package org.datanucleus.samples.jdo.tutorial;

public class Book extends Product
{
    String author=null;
    String isbn=null;
    String publisher=null;

    public Book(String name, String desc, double price, String author, String isbn,
String publisher)
    {
        super(name,desc,price);
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
    }
}
```

So we have a relationship (Inventory having a set of Products), and inheritance (Product-Book). Now we need to be able to persist objects of all of these types, so we need to **define persistence for them**. There are many things that you can define when deciding how to persist objects of a type but the essential parts are

- Mark the class as *PersistenceCapable* so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object (or use datastore-identity if no field meets this requirement).

So this is what we do now. Note that we could define persistence using XML metadata, annotations or via the JDO API. In this tutorial we will use annotations.

```
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Inventory
{
    @PrimaryKey
    String name = null;

    ...
}
```

```
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Product
{
    @PrimaryKey
    @Persistent(valueStrategy=IdGeneratorStrategy.INCREMENT)
    long id;

    ...
}
```

```
package org.datanucleus.samples.jdo.tutorial;

@PersistenceCapable
public class Book extends Product
{
    ...
}
```

Note that we mark each class that can be persisted with *@PersistenceCapable* and their primary key field(s) with @PrimaryKey. In addition we defined a *valueStrategy* for Product field *id* so that it will have its values generated automatically. In this tutorial we are using **application identity** which means that all objects of these classes will have their identity defined by the primary key field(s). You can read more in datastore identity and application identity when designing your systems persistence.

Step 2 : Define the 'persistence-unit'

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted. You do this via a file `META-INF/persistence.xml` at the root of the CLASSPATH. Like this

```
<?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">

    <!-- JDO tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jdo.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jdo.tutorial.Product</class>
        <class>org.datanucleus.samples.jdo.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <!-- Properties for runtime configuration will be added here later, see
below -->
        </properties>
    </persistence-unit>
</persistence>
```

Note that you could equally use a properties file to define the persistence with JDO, but in this tutorial we use `persistence.xml` for convenience.

Step 3 : Enhance your classes

DataNucleus JDO relies on the classes that you want to persist implementing *Persistable*. You could write your classes manually to do this but this would be laborious. Alternatively you can use a post-processing step to compilation that "enhances" your compiled classes, adding on the necessary extra methods to make them *Persistable*. There are several ways to do this, most notably at post-compile, or at runtime. We use the post-compile step in this tutorial. **DataNucleus JDO** provides its own byte-code enhancer for instrumenting/enhancing your classes (in *datanucleus-core*) and this is included in the DataNucleus AccessPlatform zip file prerequisite.

To understand on how to invoke the enhancer you need to visualise where the various source and jdo files are stored

```
src/main/java/org/datanucleus/samples/jdo/tutorial/Book.java
src/main/java/org/datanucleus/samples/jdo/tutorial/Inventory.java
src/main/java/org/datanucleus/samples/jdo/tutorial/Product.java
src/main/resources/META-INF/persistence.xml

target/classes/org/datanucleus/samples/jdo/tutorial/Book.class
target/classes/org/datanucleus/samples/jdo/tutorial/Inventory.class
target/classes/org/datanucleus/samples/jdo/tutorial/Product.class

[when using Ant]
lib/javax.jdo.jar
lib/datanucleus-core.jar
lib/datanucleus-api-jdo.jar
```

The first thing to do is compile your domain/model classes. You can do this in any way you wish, but the downloadable JAR provides an Ant task, and a Maven project to do this for you.

```
Using Ant :
ant compile

Using Maven2 :
mvn compile
```

To enhance classes using the DataNucleus Enhancer, you need to invoke a command something like this from the root of your project.

```
# Using Ant :
ant enhance

# Using Maven : (this is usually done automatically after the "compile" goal)
mvn datanucleus:enhance

# Manually on Linux/Unix :
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-api-
jdo.jar:lib/javax.jdo.jar
     org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial

# Manually on Windows :
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-api-
jdo.jar;lib\javax.jdo.jar
     org.datanucleus.enhancer.DataNucleusEnhancer -pu Tutorial

# [Command shown on many lines to aid reading - should be on single line]
```

This command enhances the .class files that have @PersistenceCapable annotations. If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a *ClassNotPersistenceCapableException* thrown. The use of the enhancer is

documented in more detail in the Enhancer Guide. The output of this step are a set of class files that represent *PersistenceCapable* classes.

Step 4 : Write the code to persist objects of your classes

Writing your own classes to be persisted is the start point, but you now need to define which objects of these classes are actually persisted, and when. Interaction with the persistence framework of JDO is performed via a PersistenceManager. This provides methods for persisting of objects, removal of objects, querying for persisted objects, etc. This section gives examples of typical scenarios encountered in an application.

The initial step is to obtain access to a PersistenceManager, which you do as follows

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("Tutorial");
PersistenceManager pm = pmf.getPersistenceManager();
```

Now that the application has a PersistenceManager it can persist objects. This is performed as follows

```
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman", "A standard discman from Sony",
49.99);
    inv.getProducts().add(product);
    pm.makePersistent(inv);
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```

Note the following

- We have persisted the *Inventory* but since this referenced the *Product* then that is also persisted.
- The *finally* step is important to tidy up any connection to the datastore, and close the PersistenceManager

If you want to retrieve an object from persistent storage, something like this will give what you

need. This uses a "Query", and retrieves all Product objects that have a price below 150.00, ordering them in ascending price order.

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

    Query q = pm.newQuery("SELECT FROM " + Product.class.getName() + " WHERE price <
150.00 ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext())
    {
        Product p = iter.next();

        ... (use the retrieved objects)
    }

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

    pm.close();
}
```

If you want to delete an object from persistence, you would perform an operation something like

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

    ... (retrieval of objects etc)

    pm.deletePersistent(product);

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

    pm.close();
}
```

Clearly you can perform a large range of operations on objects. We can't hope to show all of these here. Any good JDO book will provide many examples.

Step 5 : Run your application

To run your JDO-enabled application will require a few things to be available in the Java CLASSPATH, these being

- Any `persistence.xml` file for the PersistenceManagerFactory creation
- Any JDO XML MetaData files for your persistable classes (not used in this example)
- Any datastore driver classes (e.g JDBC driver for RDBMS, Datastax driver for Cassandra, etc) needed for accessing your datastore
- The `javax.jdo` JAR (defining the JDO interface)
- The `datanucleus-core`, `datanucleus-api-jdo` and `datanucleus-{datastore}` (for the datastore you are using, e.g *datanucleus-rdbms* when using RDBMS)

After that it is simply a question of starting your application and all should be taken care of.

In our case we need to update the `persistence.xml` with the persistence properties defining the datastore (the *properties* section of the file we showed earlier).

Firstly for RDBMS (H2 in this case)

```
<properties>
    <property name="javax.jdo.option.ConnectionURL" value="jdbc:h2:mem:nucleus1"/>
    <property name="javax.jdo.option.ConnectionDriverName" value="org.h2.Driver"/>
    <property name="javax.jdo.option.ConnectionUserName" value="sa"/>
    <property name="javax.jdo.option.ConnectionPassword" value=""/>
    <property name="datanucleus.schema.autoCreateAll" value="true"/>
</properties>
```

If we had wanted to persist to Cassandra then this would be

```
<properties>
    <property name="javax.jdo.option.ConnectionURL" value="cassandra:"/>
    <property name="javax.jdo.mapping.Schema" value="schema1"/>
    <property name="datanucleus.schema.autoCreateAll" value="true"/>
</properties>
```

or for MongoDB then this would be

```
<properties>
    <property name="javax.jdo.option.ConnectionURL" value="mongodb:/nucleus1"/>
    <property name="datanucleus.schema.autoCreateAll" value="true"/>
</properties>
```

and so on. If you look at the `persistence.xml` of the downloadable sample project it has a full range of different datastores listed to uncomment as required

You can access the DataNucleus Log file by specifying the logging configuration properties, and any messages from DataNucleus will be output in the normal way. The DataNucleus log is a very powerful way of finding problems since it can list all SQL actually sent to the datastore as well as many other parts of the persistence process.

```
# Using Ant (you need the included "persistence.xml" to specify your database)
ant run


# Using Maven:
mvn exec:java


# Manually on Linux/Unix :
java -cp lib/javax.jdo.jar:lib/datanucleus-core.jar:lib/datanucleus-{datastore}.jar:
         lib/datanucleus-api-jdo.jar:lib/{jdbc-driver}.jar:target/classes/:.
             org.datanucleus.samples.jdo.tutorial.Main


# Manually on Windows :
java -cp lib\javax.jdo.jar;lib\datanucleus-core.jar;lib\datanucleus-{datastore}.jar;
         lib\datanucleus-api-jdo.jar;lib\{jdbc-driver}.jar;target\classes\;.
             org.datanucleus.samples.jdo.tutorial.Main


Output :

DataNucleus Tutorial
=============
Persisting products
Product and Book have been persisted

Retrieving Extent for Products
>  Product : Sony Discman [A standard discman from Sony]
>  Book : JRR Tolkien - Lord of the Rings by Tolkien

Executing Query for Products with price below 150.00
>  Book : JRR Tolkien - Lord of the Rings by Tolkien

Deleting all products from persistence
Deleted 2 products

End of Tutorial
```

Step 6 : Controlling the schema

We haven't yet looked at controlling the schema generated for these classes. Now let's pay more attention to this part by defining XML Metadata for the schema. Now we will define an ORM XML metadata file to map the classes to the schema. With JDO you have various options as far as where this XML MetaData files is placed in the file structure, and whether they refer to a single class, or multiple classes in a package.

Firstly for RDBMS (H2 in this case) we define a file `package-hsql.orm` containing ORM mapping for both classes.

```
<?xml version="1.0"?>
<!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
<orm>
    <package name="org.datanucleus.samples.jdo.tutorial">
        <class name="Inventory" table="INVENTORIES">
            <field name="name">
                <column name="INVENTORY_NAME" length="100"/>
            </field>
            <field name="products" table="INVENTORY_PRODUCTS">
                <join/>
            </field>
        </class>

        <class name="Product" table="PRODUCTS">
            <inheritance strategy="new-table"/>
            <field name="id">
                <column name="PRODUCT_ID"/>
            </field>
            <field name="name">
                <column name="PRODUCT_NAME" length="100"/>
            </field>
        </class>

        <class name="Book" table="BOOKS">
            <inheritance strategy="new-table"/>
            <field name="author">
                <column length="40"/>
            </field>
            <field name="isbn">
                <column length="20" jdbc-type="CHAR"/>
            </field>
            <field name="publisher">
                <column length="40"/>
            </field>
        </class>
    </package>
</orm>
```

If we had been persisting to Cassandra then we would define a file `package-cassandra.orm` containing ORM mapping for both classes.

```
<?xml version="1.0"?>
<!DOCTYPE orm SYSTEM "file:/javax/jdo/orm.dtd">
<orm>
    <package name="org.datanucleus.samples.jdo.tutorial">
        <class name="Inventory" table="Inventories">
            <field name="name">
                <column name="Name" length="100"/>
            </field>
            <field name="products"/>
        </class>

        <class name="Product" table="Products">
            <inheritance strategy="complete-table"/>
            <field name="id">
                <column name="Id"/>
            </field>
            <field name="name">
                <column name="Name"/>
            </field>
            <field name="description">
                <column name="Description"/>
            </field>
            <field name="price">
                <column name="Price"/>
            </field>
        </class>

        <class name="Book" table="Books">
            <inheritance strategy="complete-table"/>
            <field name="Product.id">
                <column name="Id"/>
            </field>
            <field name="author">
                <column name="Author"/>
            </field>
            <field name="isbn">
                <column name="ISBN"/>
            </field>
            <field name="publisher">
                <column name="Publisher"/>
            </field>
        </class>
    </package>
</orm>
```

Again, the downloadable sample has `package-{datastore}.orm` files for many different datastores

Step 7 : Generate any schema required for your domain classes

This step is optional, depending on whether you have an existing database schema. If you haven't, at this point you can use the SchemaTool to generate the tables where these domain objects will be persisted. DataNucleus SchemaTool is a command line utility (it can be invoked from Maven/Ant in a similar way to how the Enhancer is invoked).

The first thing to do is to add an extra property to your `persistence.xml` to specify which database mapping is used (so it can locate the ORM XML metadata file).

So for H2 the *properties* section becomes

```
<properties>
    <property name="javax.jdo.option.ConnectionURL" value="jdbc:h2:mem:nucleus1"/>
    <property name="javax.jdo.option.ConnectionDriverName" value="org.h2.Driver"/>
    <property name="javax.jdo.option.ConnectionUserName" value="sa"/>
    <property name="javax.jdo.option.ConnectionPassword" value=""/>
    <property name="javax.jdo.option.Mapping" value="h2"/>
    <property name="datanucleus.schema.autoCreateAll" value="true"/>
</properties>
```

Similarly for Cassandra it would be

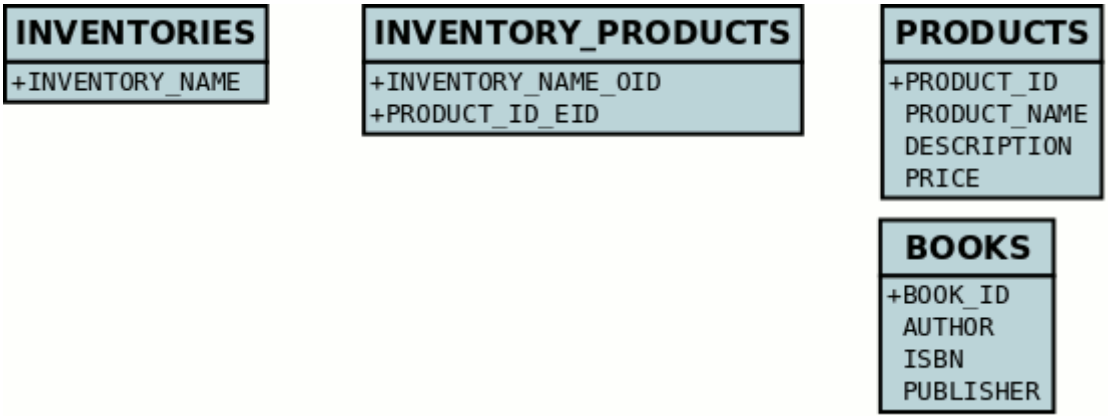
```
<properties>
    <property name="javax.jdo.option.ConnectionURL" value="cassandra:"/>
    <property name="javax.jdo.mapping.Schema" value="schema1"/>
    <property name="datanucleus.schema.autoCreateAll" value="true"/>
    <property name="javax.jdo.option.Mapping" value="cassandra"/>
</properties>
```

and so on.

Now we need to run DataNucleus SchemaTool. For our case above you would do something like this

```
# Using Ant :
ant createschema


# Using Maven2 :
mvn datanucleus:schema-create


# Manually on Linux/Unix :
java -cp target/classes:lib/datanucleus-core.jar:lib/datanucleus-
{datastore}.jar:lib/datanucleus-
javax.jdo.jar:lib/javax.jdo.jar:lib/{datastore_driver.jar}
     org.datanucleus.store.schema.SchemaTool -create -pu Tutorial

# Manually on Windows :
java -cp target\classes;lib\datanucleus-core.jar;lib\datanucleus-
{datastore}.jar;lib\datanucleus-api-
jdo.jar;lib\javax.jdo.jar;lib\{datastore_driver.jar}
     org.datanucleus.store.schema.SchemaTool -create -pu Tutorial

# [Command shown on many lines to aid reading. Should be on single line]
```

This will generate the required tables, indexes, and foreign keys for the classes defined in the JDO Meta-Data file. The generated schema (for RDBMS) in this case will be as follows



Any questions?

If you have any questions about this tutorial and how to develop applications for use with **DataNucleus** please read the online documentation since answers are to be found there. If you don't find what you're looking for go to Groups.IO or Gitter.