JPA 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 Persistence API (JPA) was designed to alleviate some of this time spent, providing an API to allow java developers to persist object-oriented data into relational databases (RDBMS).

DataNucleus JPA provides an implementation of this JPA *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 JPA 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 JPA for java persistence.

- Your classes should be exactly that, *your* classes. DataNucleus imposes little to nothing on you. Some JPA providers insist on a *default constructor*, but DataNucleus provides its enhancer to add that automatically when not present.
- Your JPA entity 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 JPA Mapping Guide
- Use of JPA requires an EntityManagerFactory to access the datastore.
- The persistence itself is controlled by an EntityManager 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 JPA you can use JPQL or SQL query languages
- You will need `javax.persistence` as well as `datanucleus-api-jpa`, `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.persistence.jar` : This is the JPA API. This is basically a collection of interfaces, annotations and helper classes.
- `datanucleus-api-jpa.jar` : This is DataNucleus' implementation of the JPA API. It implements the interfaces defined in *javax.persistence.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-jpa-query.jar` : This provides an *annotation processor* and is used by the JPA Criteria query mechanism to generate the static metamodel 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).

JPA Tutorial (v5.0)

Download Source Code (GitHub)

Background

An application can be JPA-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 Dali Eclipse plugin coupled with 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 JPA 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-jpa-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 (in this case RDBMS). You can do this from the 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.jpa.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.jpa.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.jpa.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 an *Entity* so it is visible to the persistence mechanism
- Identify which field(s) represent the identity of the object.

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

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

@Entity
public class Inventory
{
    @Id
    String name = null;

    @OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH})
    Set<Product> products = new HashSet();
    ...
}
```

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

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Product
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    long id;

    ...
}
```

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

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

Note that we mark each class that can be persisted with `@Entity` and their primary key field(s) with `@Id`. 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 the application identity guide 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">

    <!-- JPA tutorial "unit" -->
    <persistence-unit name="Tutorial">
        <class>org.datanucleus.samples.jpa.tutorial.Inventory</class>
        <class>org.datanucleus.samples.jpa.tutorial.Product</class>
        <class>org.datanucleus.samples.jpa.tutorial.Book</class>
        <exclude-unlisted-classes/>
        <properties>
            <!-- For adding runtime properties. See later -->
        </properties>
    </persistence-unit>
</persistence>
```

Step 3 : Enhance your classes

DataNucleus relies on the classes that you want to persist be enhanced to implement the interface *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 JPA** provides its own byte-code enhancer for instrumenting/enhancing your classes (in `datanucleus-core.jar`) 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 metadata files are stored

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

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

# when using Ant
lib/javax.persistence.jar
lib/datanucleus-core.jar
lib/datanucleus-api-jpa.jar
```

The first thing to do is compile your domain/model classes. You can do this in any way you wish, but

the download provides an Ant task, and a Maven project to do this for you.

```
# Using Ant :
ant compile

# Using Maven :
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-
jpa.jar:lib/javax.persistence.jar
     org.datanucleus.enhancer.DataNucleusEnhancer -api JPA -pu Tutorial

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

This command enhances all classes defined in the persistence-unit "Tutorial". If you accidentally omitted this step, at the point of running your application and trying to persist an object, you would get a *ClassNotPersistableException* 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 persistable 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 JPA is performed via an EntityManager. 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 an EntityManager, which you do as follows

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Tutorial");
EntityManager em = emf.createEntityManager();
```

So we created an *EntityManagerFactory* for our "persistence-unit" called "Tutorial" which we defined above. Now that the application has an EntityManager it can persist objects. This is performed as follows

```
Transaction tx = em.getTransaction();
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);
    em.persist(inv);

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

    em.close();
}
```

Please note that the *finally* step is important in that it tidies up connections to the datastore and the EntityManager. Now we want to retrieve some objects from persistent storage, so we will use a "Query". In our case we want access to all Product objects that have a price below 150.00 and ordering them in ascending order.

```
Transaction tx = em.getTransaction();
try
{
    tx.begin();

    Query q = pm.createQuery("SELECT p FROM Product p WHERE p.price < 150.00");
    List results = q.getResultList();
    Iterator iter = results.iterator();
    while (iter.hasNext())
    {
        Product p = (Product)iter.next();

        ... (use the retrieved object)
    }

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

    em.close();
}
```

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

```
Transaction tx = em.getTransaction();
try
{
    tx.begin();

    // Find and delete all objects whose last name is 'Jones'
    Query q = em.createQuery("DELETE FROM Person p WHERE p.lastName = 'Jones'");
    int numberInstancesDeleted = q.executeUpdate();

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

    em.close();
}
```

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

Step 5 : Run your application

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

- The `persistence.xml` file (stored under META-INF/)
- Any ORM MetaData files for your persistable classes
- Any Datastore driver classes (e.g JDBC driver for RDBMS, Datastax driver for Cassandra, etc) needed for accessing your datastore
- The `javax.persistence.jar` (defining the JPA interface)
- The `datanucleus-core.jar`, `datanucleus-api-jpa.jar` and `datanucleus-{datastore}.jar` (for the datastore you are using, e.g *datanucleus-rdbms.jar* 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 firstly need to update the `persistence.xml` with the persistence properties defining the datastore (the *properties* section of the file we showed earlier), like this

```
<properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:nucleus1"/>
    <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
    <property name="javax.persistence.jdbc.user" value="sa"/>
    <property name="javax.persistence.jdbc.password" 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.persistence.jdbc.url" value="cassandra:"/>
    <property name="datanucleus.mapping.Schema" value="schema1"/>
    <property name="datanucleus.schema.autoCreateAll" value="true"/>
</properties>
```

or for MongoDB then this would be

```
<properties>
    <property name="javax.persistence.jdbc.url" 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 etc 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.persistence.jar:lib/datanucleus-core.jar:lib/datanucleus-
rdbms.jar:lib/datanucleus-api-jpa.jar:lib/{datastore-driver}.jar:target/classes/:.
     org.datanucleus.samples.jpa.tutorial.Main


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


# Output :

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

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

Deleting all products from persistence

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. In this example we define this in XML to separate schema information from persistence information (though could equally have used annotations if we really wanted to). This information is used *either* to match up to an existing schema, *or* is used to generate a new schema (see #Step 7). So we define a file `META-INF/orm.xml` at the root of the CLASSPATH. Like this

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings>
    <description>DataNucleus JPA tutorial</description>
    <package>org.datanucleus.samples.jpa.tutorial</package>
    <entity class="org.datanucleus.samples.jpa.tutorial.Product" name="Product">
        <table name="JPA_PRODUCTS"/>
        <attributes>
            <id name="id">
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="name">
                <column name="PRODUCT_NAME" length="100"/>
            </basic>
            <basic name="description">
                <column length="255"/>
            </basic>
        </attributes>
    </entity>

    <entity class="org.datanucleus.samples.jpa.tutorial.Book" name="Book">
        <table name="JPA_BOOKS"/>
        <attributes>
            <basic name="isbn">
                <column name="ISBN" length="20"></column>
            </basic>
            <basic name="author">
                <column name="AUTHOR" length="40"/>
            </basic>
            <basic name="publisher">
                <column name="PUBLISHER" length="40"/>
            </basic>
        </attributes>
    </entity>

    <entity class="org.datanucleus.samples.jpa.tutorial.Inventory" name="Inventory">
        <table name="JPA_INVENTORY"/>
        <attributes>
            <id name="name">
                <column name="NAME" length="40"></column>
            </id>
            <one-to-many name="products">
                <join-table name="JPA_INVENTORY_PRODUCTS">
                    <join-column name="INVENTORY_ID_OID"/>
                    <inverse-join-column name="PRODUCT_ID_EID"/>
                </join-table>
            </one-to-many>
        </attributes>
    </entity>
</entity-mappings>
```

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 add the property **javax.persistence.schema-generation.database.action** to your `persistence.xml` and set it to *create* and this will create the schema for the specified classes when the *EntityManagerFactory* is created. The first thing that you need is to update the `src/main/resources/META-INF/persistence.xml` file with your database details, and this property.

For H2 these properties become

```
<properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:nucleus1"/>
    <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
    <property name="javax.persistence.jdbc.user" value="sa"/>
    <property name="javax.persistence.jdbc.password" value=""/>

    <property name="javax.persistence.schema-generation.database.action"
value="create/>
</properties>
```

For other datastores, just look at the downloadable sample and uncomment as required.

Now we simply create the EntityManagerFactory as earlier. This will generate the required tables, indexes, and foreign keys for the classes defined in the annotations and `orm.xml` Meta-Data file.

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