DataNucleus Object Store

Table of Contents

Chapter 1. DataNucleus Object Store

The DataNucleus Object Store enables domain objects to be persisted to relational as well as NoSQL databases. The object store is implemented using DataNucleus.

This user guide discuss end-user features, configuration and customization of the DataNucleus object store.

DataNucleus as a product also supports the JPA API; Apache Isis is likely to also support JPA in the future.

1.1. Other Guides

Apache Isis documentation is broken out into a number of user and reference guides.

The user guides available are:

- Fundamentals
- Wicket viewer
- Restful Objects viewer
- DataNucleus object store (this guide)
- Security
- Testing
- Beyond the Basics

The reference guides are:

- Annotations
- Domain Services
- Configuration Properties
- Classes, Methods and Schema
- Apache Isis Maven plugin
- Framework Internal Services

The remaining guides are:

- Developers' Guide (how to set up a development environment for Apache Isis and contribute back to the project)
- Committers' Guide (release procedures and related practices)

Chapter 2. Configuring DataNucleus

Apache Isis programmatically configures DataNucleus; any Apache Isis properties with the prefix `isis.persistor.datanucleus.impl` are passed through directly to the JDO/DataNucleus objectstore (with the prefix stripped off, of course).

DataNucleus will for itself also and read the `META-INF/persistence.xml`; at a minimum this defines the name of the "persistence unit". n theory it could also hold mappings, though in Apache Isis we tend to use annotations instead.

Furthermore, DataNucleus will search for various other XML mapping files, eg `mappings.jdo`. A full list can be found here. The metadata in these XML can be used to override the annotations of annotated entities; see Overriding JDO Annotatons for further discussion.

2.1. Configuration Properties

These configuration properties are typically stored in `WEB-INF/persistor_datanucleus.properties`. However, you can place all configuration properties into `WEB-INF/isis.properties` if you wish (the configuration properties from all config files are merged together).

2.1.1. Configuration Properties for Apache Isis itself

Table 1. JDO/DataNucleus Objectstore Configuration Properties

Property	**Value** (*default value*)	**Description**
`isis.persistor.datanucleus.classMetadataLoadedListener`	`FQCN`	The default (`o.a.i.os.jdo.dn.CreateSchemaObjectFromClassMetadata`) creates a DB schema object
`isis.persistor.datanucleus.RegisterEntities.packagePrefix`	fully qualified package names (CSV)	that specifies the entities early rather than allow DataNucleus to find the entities lazily. Further discussion below. This property is IGNORED if the `isis.appManifest` configuration property is specified, or if an `AppManifest` is provided programmatically.
`isis.persistor.datanucleus.PublishingService.serializedForm`	zipped	

2.1.2. Configuration Properties passed through directly to DataNucleus.

Table 2. JDO/DataNucleus Objectstore Configuration Properties

Property	**Value** (*default value*)	**Description**
`isis.persistor.datanucleus.impl.*`		Passed through directly to Datanucleus (with `isis.persistor.datanucleus.impl` prefix stripped)

Property	Value (*default value*)	Description
`isis.persistor.datanucleus.impl.datanucleus.persistenceByReachabilityAtCommit`	`false`	We recommend this setting is disabled. Further discussion below.

2.2. `persistence.xml`

TODO

2.3. Eagerly Registering Entities

Both Apache Isis and DataNucleus have their own metamodels of the domain entities. Apache Isis builds its metamodel by walking the graph of types of the domain services. The JDO/DataNucleus objectstore then takes these types and registers them with DataNucleus.

In some cases, though, not every entity type is discoverable from the API of the service actions. This is especially the case if you have lots of subtypes (where the action method specifies only the supertype). In such cases the Isis and JDO metamodels is built lazily, when an instance of that (sub)type is first encountered.

Apache Isis is quite happy for the metamodel to be lazily created, and - to be fair - DataNucleus also works well in most cases. In some cases, though, we have found that the JDBC driver (eg HSQLDB) will deadlock if DataNucleus tries to submit some DDL (for a lazily discovered type) intermingled with DML (for updating). In any case, it's probably not good practice to have DataNucleus work this way.

The framework thus provide mechanisms to search for all `@PersistenceCapable` entities under specified package(s), and registers them all eagerly. In fact there are two:

- as of 1.9.0 the recommended (and simpler) approach is to specify an `AppManifest`, either as a `isis.appManifest` configuration property or programmatically.
- for earlier versions the `isis.persistor.datanucleus.RegisterEntities.packagePrefix` configuration property can be specified. To bootstrap as a webapp this is usually specified in `persistor_datanucleus.properties`. (This is also supported in 1.9.0 if no `AppManifest` is specified. For integration testing this can be specified programatically.

Further discussion on specifying the package(s) in integration testing (for either approach) can be found in the user guide.

2.4. Persistence by Reachability

By default, JDO/DataNucleus supports the concept of persistence-by-reachability. That is, if a non-persistent entity is associated with an already-persistent entity, then DataNucleus will detect this and will automatically persist the associated object. Put another way: there is no need to call Apache Isis' `DomainObjectContainer#persist(.)` or `DomainObjectContainer#persistIfNotAlready(.)`

methods.

However, convenient though this feature is, you may find that it causes performance issues.

DataNucleus' persistence-by-reachability may cause performance issues. We strongly recommend that you disable it.

One scenario in particular where this performance issues can arise is if your entities implement the `java.lang.Comparable` interface, and you have used Apache Isis' `ObjectContracts` utility class. The issue here is that `ObjectContracts` implementation can cause DataNucleus to recursively rehydrate a larger number of associated entities. (More detail below).

We therefore recommend that you disable persistence-by-reachability by adding the following to `persistor_datanucleus.properties`:

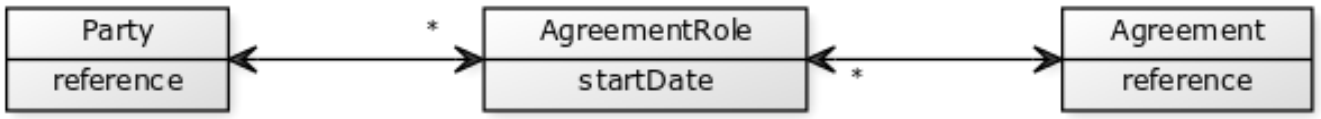
```
isis.persistor.datanucleus.impl.datanucleus.persistenceByReachabilityAtCommit=false
```

This change has been made to the SimpleApp archetype

If you do disable this feature, then you will (of course) need to ensure that you explicitly persist all entities using the `DomainObjectContainer#persist(.)` or `DomainObjectContainer#persistIfNotAlready(.)` methods.

2.4.1. The issue in more detail

Consider these entities (yuml.me/b8681268):



In the course of a transaction, the `Agreement` entity is loaded into memory (not necessarily modified), and then new `AgreementRole`s are associated to it.

All these entities implement `Comparable` using `ObjectContracts`, and the implementation of `AgreementRole`'s (simplified) is:

```
public class AgreementRole {
    ...
    public int compareTo(AgreementRole other) {
        return ObjectContracts.compareTo(this, other, "agreement","startDate","party"
);
    }
}
```

while `Agreement`'s is implemented as:

```
public class Agreement {
    ...
    public int compareTo(Agreement other) {
        return ObjectContracts.compareTo(this, other, "reference");
    }
}
```

and `Party`'s is similarly implemented as:

```
public class Party {
    ...
    public int compareTo(Party other) {
        return ObjectContracts.compareTo(this, other, "reference");
    }
}
```

DataNucleus's persistence-by-reachability algorithm adds the `AgreementRole` instances into a `SortedSet`, which causes `AgreementRole#compareTo()` to fire:

- the evaluation of the "agreement" property delegates back to the `Agreement`, whose own `Agreement#compareTo()` uses the scalar `reference` property. As the `Agreement` is already in-memory, this does not trigger any further database queries
- the evaluation of the "startDate" property is just a scalar property of the `AgreementRole`, so will already in-memory
- the evaluation of the "party" property delegates back to the `Party`, whose own `Party#compareTo()` requires the uses the scalar `reference` property. However, since the `Party` is not yet in-memory, using the `reference` property triggers a database query to "rehydrate" the `Party` instance.

In other words, in figuring out whether `AgreementRole` requires the persistence-by-reachability algorithm to run, it causes the adjacent associated entity `Party` to also be retrieved.

2.5. Using JNDI DataSource

Isis' JDO objectstore can be configured either to connect to the database using its own connection pool, or by using a container-managed datasource.

2.5.1. Application managed

Using a connection pool managed directly by the application (that is, by Apache Isis' JDO objectstore and ultimately by DataNucleus) requires a single set of configuration properties to be specified.

In either `WEB-INF\isis.properties` file (or `WEB-INF\persistor.properties`, or `WEB-INF\persistor_datanucleus.properties`), specify the connection driver, url, username and password.

For example:

```
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionDriverName=net.sf.log4jdbc.
DriverSpy
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionURL=jdbc:log4jdbc:hsqldb:me
m:test
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionUserName=sa
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionPassword=
```

Those configuration properties that start with the prefix `isis.persistor.datanucleus.impl.` are passed through directly to DataNucleus (with the prefix removed).

It is also possible to specify the ` datanucleus.ConnectionPasswordDecrypter ` property; see the DataNucleus documentation for further details.

2.5.2. Container managed (JNDI)

Using a datasource managed by the servlet container requires three separate bits of configuration.

Firstly, specify the name of the datasource in the `WEB-INF\persistor_datanucleus.properties` file. For example:

If connection pool settings are also present in this file, they will simply be ignored. Any other configuration properties that start with the prefix `isis.persistor.datanucleus.impl.` are passed through directly to DataNucleus (with the prefix removed).

Secondly, in the `WEB-INF/web.xml`, declare the resource reference:

```
<resource-ref>
    <description>db</description>
    <res-ref-name>jdbc/simpleapp</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

Finally, declare the datasource as required by the servlet container. For example, if using Tomcat 7, the datasource can be specified by adding the following to `$TOMCAT_HOME/conf/context.xml`:

```
<Resource name="jdbc/simpleapp"
  auth="Container"
  type="javax.sql.DataSource"
  maxActive="100"
  maxIdle="30"
  maxWait="10000"
  username="sa"
  password="p4ssword"
  driverClassName="com.microsoft.sqlserver.jdbc.SQLServerDriver"
  url="jdbc:sqlserver://127.0.0.1:1433;instance=.;databaseName=simpleapp"/>
```

You will also need to make sure that the JDBC driver is on the servlet container's classpath. For Tomcat, this means copying the driver to `$TOMCAT_HOME/lib`.

According to Tomcat's documentation, it is supposedly possible to copy the `conf/context.xml` to the name of the webapp, eg `conf/mywebapp.xml`, and scope the connection to that webapp only. I was unable to get this working, however.

Chapter 3. JDO Mappings

3.1. 1-m Bidirectional relationships

Consider a bidirectional one-to-many association between two entities; a collection member in the "parent" and a property member on the "child".

We can tell DataNucleus about the bidirectionality using `@Persistent(mappedBy=…)`, or we can take responsibility for this aspect ourselves.

In addition, the two entities can be associated either without or with a join table (indicated by the `@Join` annotation):

- without a join table is more common; a regular foreign key in the child table for `FermentationVessel` points back up to the associated parent `Batch`
- with a join table; a link table holds the tuple representing the linkage.

Testing (as of `1.13.0`, against `dn-core 4.1.7`/`dn-rdbms 4.1.9`) has determined there are two main rules:

- If not using `@Join`, then the association must be maintained by setting the child association on the parent.

  It is not sufficient to simply add the child object to the parent's collection.

- `@Persistent(mappedBy=…)` and `@Join` cannot be used together.

  Put another way, if using `@Join` then you must maintain both sides of the relationship in the application code.

In the examples that follow, we use two entities, `Batch` and `FermentationVessel` (from a brewery domain). In the original example domain the relationship between these two entities was optional (a `FermentationVessel` may have either none or one `Batch` associated with it); for the purpose of this article we'll explore both mandatory and optional associations.

3.1.1. Mandatory, no `@Join`

In the first scenario we have use `@Persistent(mappedBy=…)` to indicate a bidirectional association, without any `@Join`:

```
public class Batch {

    // getters and setters omitted

    @Persistent(mappedBy = "batch", dependentElement = "false")     ①
    private SortedSet<FermentationVessel> vessels = new TreeSet<FermentationVessel>();
}
```

① "mappedBy" means this is bidirectional

and

```
public class FermentationVessel implements Comparable<FermentationVessel> {

    // getters and setters omitted

    @Column(allowsNull = "false")       ①
    private Batch batch;

    @Column(allowsNull = "false")
    private State state;                ②
}
```

① mandatory association up to parent

② State is an enum (omitted)

Which creates this schema:

```
CREATE TABLE "batch"."Batch"
(
    "id" BIGINT GENERATED BY DEFAULT AS IDENTITY,
    ...
    "version" BIGINT NOT NULL,
    CONSTRAINT "Batch_PK" PRIMARY KEY ("id")
)
CREATE TABLE "fvessel"."FermentationVessel"
(
    "id" BIGINT GENERATED BY DEFAULT AS IDENTITY,
    "batch_id_OID" BIGINT NOT NULL,
    "state" NVARCHAR(255) NOT NULL,
    ...
    "version" TIMESTAMP NOT NULL,
    CONSTRAINT "FermentationVessel_PK" PRIMARY KEY ("id")
)
```

That is, there is an mandatory foreign key from `FermentationVessel` to `Batch`.

In this case we can use this code:

```
public Batch transfer(final FermentationVessel vessel) {
    vessel.setBatch(this);                                  ①
    vessel.setState(FermentationVessel.State.FERMENTING);
    return this;
}
```

① set the parent on the child

This sets up the association correctly, using this SQL:

```
UPDATE "fvessel"."FermentationVessel"
   SET "batch_id_OID"=<0>
       ,"state"=<'FERMENTING'>
       ,"version"=<2016-07-07 12:37:14.968>
 WHERE "id"=<0>
```

The following code will also work:

```
public Batch transfer(final FermentationVessel vessel) {
    vessel.setBatch(this);                                  ①
    getVessels().add(vessel);                               ②
    vessel.setState(FermentationVessel.State.FERMENTING);
    return this;
}
```

① set the parent on the child

② add the child to the parent's collection.

However, obviously the second statement is redundant.

3.1.2. Optional, no `@Join`

If the association to the parent is made optional:

```
public class FermentationVessel implements Comparable<FermentationVessel> {

    // getters and setters omitted

    @Column(allowsNull = "true")       ①
    private Batch batch;

    @Column(allowsNull = "false")
    private State state;
}
```

① optional association up to parent

Which creates this schema:

```
CREATE TABLE "batch"."Batch"
(
    "id" BIGINT GENERATED BY DEFAULT AS IDENTITY,
    ...
    "version" BIGINT NOT NULL,
    CONSTRAINT "Batch_PK" PRIMARY KEY ("id")
)
CREATE TABLE "fvessel"."FermentationVessel"
(
    "id" BIGINT GENERATED BY DEFAULT AS IDENTITY,
    "batch_id_OID" BIGINT NULL,
    "state" NVARCHAR(255) NOT NULL,
    ...
    "version" TIMESTAMP NOT NULL,
    CONSTRAINT "FermentationVessel_PK" PRIMARY KEY ("id")
)
```

This is almost exactly the same, except the foreign key from `FermentationVessel` to `Batch` is now nullable.

In this case then setting the parent on the child still works:

```
public Batch transfer(final FermentationVessel vessel) {
    vessel.setBatch(this);                                  ①
    vessel.setState(FermentationVessel.State.FERMENTING);
    return this;
}
```

① set the parent on the child

HOWEVER, if we (redundantly) update both sides, then - paradoxically - the association is NOT set up

```
public Batch transfer(final FermentationVessel vessel) {
    vessel.setBatch(this);                                  ①
    getVessels().add(vessel);                               ②
    vessel.setState(FermentationVessel.State.FERMENTING);
    return this;
}
```

① set the parent on the child

② add the child to the parent's collection.

It's not clear if this is a bug in `dn-core 4.1.7`/`dn-rdbms 4.19`; an earlier thread on the mailing list from 2014 actually gave the opposite advice, see this thread and in particular this message.

In fact we also have a different case which argues that the parent should only be set on the child, and the child *not* added to the parent's collection. This concurs with the most recent testing.

Therefore, the simple advice is that, for bidirectional associations, simply set the parent on the child, and this will work reliably irrespective of whether the association is mandatory or optional.

3.1.3. With `@Join`

Although DataNucleus does not complain if `@Persistence(mappedBy=…)` and `@Join` are combined, testing (against `dn-core 4.1.7`/`dn-rdbms 4.19`) has shown that the bidirectional association is not properly maintained.

Therefore, we recommend that if `@Join` is used, then manually maintain both sides of the relationship and do not indicate that the association is bidirectional.

For example:

```
public class Batch {

    // getters and setters omitted

    @Join(table = "Batch_vessels")
    @Persistent(dependentElement = "false")
    private SortedSet<FermentationVessel> vessels = new TreeSet<FermentationVessel>();
}
```

and

```
public class FermentationVessel implements Comparable<FermentationVessel> {

    // getters and setters omitted

    @Column(allowsNull = "true")       ①
    private Batch batch;

    @Column(allowsNull = "false")
    private State state;
}
```

① optional association up to parent

creates this schema:

```
CREATE TABLE "batch"."Batch"
(
    "id" BIGINT GENERATED BY DEFAULT AS IDENTITY,
    ...
    "version" BIGINT NOT NULL,
    CONSTRAINT "Batch_PK" PRIMARY KEY ("id")
)
CREATE TABLE "fvessel"."FermentationVessel"
(
    "id" BIGINT GENERATED BY DEFAULT AS IDENTITY,
    "state" NVARCHAR(255) NOT NULL,
    ...
    "version" TIMESTAMP NOT NULL,
    CONSTRAINT "FermentationVessel_PK" PRIMARY KEY ("id")
)
CREATE TABLE "batch"."Batch_vessels"
(
    "id_OID" BIGINT NOT NULL,
    "id_EID" BIGINT NOT NULL,
    CONSTRAINT "Batch_vessels_PK" PRIMARY KEY ("id_OID","id_EID")
)
```

That is, there is NO foreign key from `FermentationVessel` to `Batch`, instead the `Batch_vessels` table links the two together.

These should then be maintained using:

```
public Batch transfer(final FermentationVessel vessel) {
    vessel.setBatch(this);                                  ①
    getVessels().add(vessel);                               ②
    vessel.setState(FermentationVessel.State.FERMENTING);
    return this;
}
```

① set the parent on the child

② add the child to the parent's collection.

that is, explicitly update both sides of the relationship.

This generates this SQL:

```
INSERT INTO "batch"."Batch_vessels" ("id_OID","id_EID") VALUES (<0>,<0>)
UPDATE "batch"."Batch"
   SET "version"=<3>
 WHERE "id"=<0>
UPDATE "fvessel"."FermentationVessel"
   SET "state"=<'FERMENTING'>
      ,"version"=<2016-07-07 12:49:21.49>
 WHERE "id"=<0>
```

It doesn't matter in these cases whether the association is mandatory or optional; it will be the same SQL generated.

Chapter 4. Overriding JDO Annotations

The JDO Objectstore (or rather, the underlying DataNucleus implementation) builds its own persistence metamodel by reading both annotations on the class and also by searching for metadata in XML files. The metadata in the XML files takes precedence over the annotations, and so can be used to override metadata that is "hard-coded" in annotations.

For example, as of 1.9.0 the various Isis addons modules (not ASF) use schemas for each entity. For example, the `AuditEntry` entity in the audit module is annotated as:

```
@javax.jdo.annotations.PersistenceCapable(
        identityType=IdentityType.DATASTORE,
        schema = "IsisAddonsAudit",
        table="AuditEntry")
public class AuditEntry {
    ...
}
```

This will map the `AuditEntry` class to a table `"IsisAddonsAudit"."AuditEntry"`; that is using a custom schema to own the object.

Suppose though that for whatever reason we didn't want to use a custom schema but would rather use the default. We can override the above annotation using a `package.jdo` file, for example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<jdo xmlns="http://xmlns.jcp.org/xml/ns/jdo/jdo"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/jdo
        http://xmlns.jcp.org/xml/ns/jdo/jdo_3_0.xsd" version="3.0">
    <package name="org.isisaddons.module.audit.dom">
        <class name="AuditEntry" schema="PUBLIC" table="IsisAddonsAuditEntry">
        </class>
    </package>
</jdo>
```

This file should be placed can be placed in `src/main/java/META-INF` within your application's `dom` module.

You can use a mixin action on `Persistable` mixin to download the JDO class metadata in XML form.

- The same approach should work for any other JDO metadata, but some experimentation might be required.+

  For example, in writing up the above example we found that writing `schema=""` (in an attempt to say, "use the default schema for this table") actually caused the original annotation value to be used instead.

- Forcing the schema to "PUBLIC" (as in the above example) works, but it isn't ideal because the name "PUBLIC" is not vendor-neutral (it works for HSQLDB, but MS SQL Server uses "dbo" as its default).
- As of 1.9.0 Apache Isis will automatically (attempt) to create the owning schema for a given table if it does not exist. This behaviour can be customized, as described in the section on using modules.
- You may need to override the entire class metadata rather than individual elements; the mixin mentioned above can help here.

Chapter 5. Java8

DataNucleus 4.x supports Java 7, but can also be used with Java 8, eg for streams support against collections managed by DataNucleus.

Just include within `<dependencies>` of your `dom` module's `pom.xml`:

```
<dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-java8</artifactId>
    <version>4.2.0-release</version>t
</dependency>
```

The DataNucleus website includes a page listing version compatibility of these extensions vis-a-vis the core DataNucleus platform.