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". In 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.standaloneCollection.bulkLoad`	`true`, `false` (`false`)	Enables bulk load of standalone collections. Further discussion below.
`isis.persistor.datanucleus.classMetadataLoadedListener`	fully qualified class name (`o.a.i.os.jdo.dn.CreateSchemaObjectFromClassMetadata`)	The default implementation creates a DB schema object. There generally is no need to change this from its default.
`isis.persistor.datanucleus.RegisterEntities.packagePrefix`	fully qualified package names, CSV	This property is derived automatically derived from the set of modules provided in the `AppManifest`, and so does not need to be specified explicitly. It holds the set of packages to search so that DataNucleus builds its metamodel eagerly rather than lazily.

Property	Value (*default value*)	Description
`isis.persistor.datanucleus.PublishingService.serializedForm`	zipped	(`PublishingService` is deprecated, and therefore so is this property).

Also:

Property	Value (default value)	Description
`isis.persistor.disable ConcurrencyChecking`	`true`,`false` (`false`)	Disables concurrency checking globally. Only intended for "emergency use" as a workaround while pending fix/patch to Apache Isis itself. (Note that there is no "datanucleus" in the property).

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)
`isis.persistor.datanucleus.impl.datanucleus.persistenceByReachabilityAtCommit`	`false`	We recommend this setting is disabled. Further discussion below.

2.2. Bulk Load of Standalone Collections

The implementation of user interactions (meaning either an action invocations or a property edits) in the Wicket viewer is splits into two. The first phase performs the actual interaction, with the results (dirtied objects) flushed to the database. The second phase then renders the results of the interaction.

When the user interaction in question is an action invocation that returns a list of objects, the resultant list is not rendered in the first phase. Instead, only the IDs of the objects in the list are captured. When the list is then rendered, the framework re-loads each object.

The default implementation does this row-by-row, resulting in multiple queries against the database. Setting the property:

```
isis.persistor.datanucleus.standaloneCollection.bulkLoad=true
```

changes to a more efficient implementation that bulk loads all the objects using a single query.

In the future the bulkLoad implementation may be made the default.

The implementation of parented collections does not suffer from this issue; the rendering phase runs the query to obtain the matches.

2.3. 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' `RepositoryService#persist(.)` or `RepositoryService#persistAndFlush(.)` 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 both the HelloWorld and SimpleApp archetypes.

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

2.3.1. 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.

2.4. `persistence.xml`

DataNucleus will for itself also and read the `META-INF/persistence.xml`. In theory it can hold mappings and even connection strings. However, with Apache Isis we tend to use annotations

instead and externalize connection strings. so its definition is extremely simply, specifying just the name of the "persistence unit".

Here's the one provided by the SimpleApp archetype:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <persistence-unit name="simple">
    </persistence-unit>
</persistence>
```

Normally all one needs to do is to change the `persistence-unit` name.

If you use Eclipse IDE on Windows then note the importance of the `persistence.xml` file to make DataNucleus enhancer work correctly.

See DataNucleus' documentation on `persistence.xml` to learn more.

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 (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 had a different case raised (url lost) 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.

3.2. Mandatory Properties in Subtypes

If you have a hierarchy of classes then you need to decide which inheritance strategy to use.

- "table per hierarchy", or "rollup" (`InheritanceStrategy.SUPERCLASS_TABLE`)

  whereby a single table corresponds to the superclass, and also holds the properties of the subtype (or subtypes) being rolled up

- "table per class" (`InheritanceStrategy.NEW_TABLE`)

  whereby there is a table for both superclass and subclass, in 1:1 correspondence

- "rolldown" (`InheritanceStrategy.SUBCLASS_TABLE`)

  whereby a single table holds the properties of the subtype, and also holds the properties of its supertype

In the first "rollup" case, we can have a situation where - logically speaking - the property is mandatory in the subtype - but it must be mapped as nullable in the database because it is n/a for any other subtypes that are rolled up.

In this situation we must tell JDO that the column is optional, but to Apache Isis we want to enforce it being mandatory. This can be done using the `@Property(optionality=Optionality.MANDATORY)` annotation.

For example:

```
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.SUPER_TABLE)
public class SomeSubtype extends SomeSuperType {
    @javax.jdo.annotations.Column(allowsNull="true")
    @Property(optionality=Optionality.MANDATORY)
    @lombok.Getter @lombok.Setter
    private LocalDate date;
}
```

The `@Property(optionality=…)` annotation is equivalent to the older but still supported `@Optional` annotation and `@Mandatory` annotations.

3.3. Mapping to a View

JDO/DataNucleus supports the ability to map the entity that is mapped to a view rather than a database table. Moreover, DataNucleus itself can create/maintain this view.

One use case for this is to support use cases which act upon aggregate information. An example is in the (non-ASF) Estatio application, which uses a view to define an "invoice run": a representatoin of all pending invoices to be sent out for a particular shopping centre. (Note that example also shows the entity as being "non-durable", but if the view is read/write then — I think — that this isn't necessary required).

For more on this topic, see the DataNucleus documentation.

Chapter 4. Database Schemas

In the same way that Java packages act as a namespace for domain objects, it's good practice to map domain entities to their own (database) schemas.

All the (non-ASF) Incode Platform modules do this. For example:

```
@javax.jdo.annotations.PersistenceCapable( ...
        schema = "isissecurity",
        table = "ApplicationUser")
public class ApplicationUser ... { ... }
```

results in a `CREATE TABLE` statement of:

```
CREATE TABLE isissecurity."ApplicationUser" (
    ...
)
```

while:

```
@javax.jdo.annotations.PersistenceCapable( ...
        schema = "isisaudit",
        table="AuditEntry")
public class AuditEntry ... { ... }
```

similarly results in:

```
CREATE TABLE isisaudit."AuditEntry" (
    ...
)
```

If for some reason you don't want to use schemas (though we strongly recommend that you do), then note that you can override the `@PersistenceCapable` annotation by providing XML metadata (the `mappings.jdo` file). See the section on configuring DataNucleus Overriding Annotations for more details.

4.1. Listener to create schema

Apache Isis automatically creates owning schema objects for the tables that correspond to each entity class in the JDO metamodel.

This is done by installing a listener, `CreateSchemaObjectFromClassMetadata`, on a callback provided by JDO/DataNucleus. The listener is invoked on the initialization of each class. It checks for the schema's existence, and creates the schema if required.

The guts of its implementation is:

```
public class CreateSchemaObjectFromClassMetadata
        implements MetaDataListener,
                   DataNucleusPropertiesAware {
    @Override
    public void loaded(final AbstractClassMetaData cmd) { ... }

    protected String buildSqlToCheck(final AbstractClassMetaData cmd) {
        final String schemaName = schemaNameFor(cmd);
        return String.format(
            "SELECT count(*) FROM INFORMATION_SCHEMA.SCHEMATA where SCHEMA_NAME =
'%s'", schemaName);
    }
    protected String buildSqlToExec(final AbstractClassMetaData cmd) {
        final String schemaName = schemaNameFor(cmd);
        return String.format("CREATE SCHEMA \"%s\"", schemaName);
    }
}
```

where `MetaDataListener` is the DataNucleus listener API:

```
public interface MetaDataListener {
    void loaded(AbstractClassMetaData cmd);
}
```

Although not formal API, the default `CreateSchemaObjectFromClassMetadata` has been designed to be easily overrideable if you need to tweak it to support other RDBMS'. Any implementation must implement `org.datanucleus.metadata.MetaDataListener`:

The implementation provided has has been tested for HSQLDB, PostgreSQL and MS SQL Server, and is used automatically unless an alternative implementation is specified (as described in the section below).

4.2. Alternative implementation

An alternative implementation can be registered and used through the following configuration property:

```
isis.persistor.datanucleus.classMetadataLoadedListener=\

org.apache.isis.objectstore.jdo.datanucleus.CreateSchemaObjectFromClassMetadata
```

Because this pertains to the JDO Objectstore we suggest you put this configuration property in `WEB-INF/persistor_datanucleus.properties`; but putting it in `isis.properties` will also work.

Any implementation must implement `org.datanucleus.metadata.MetaDataListener`. In many cases

simply subclassing from `CreateSchemaObjectFromClassMetadata` and overriding `buildSqlToCheck(…)` and `buildSqlToExec(…)` should suffice.

If you *do* need more control, your implementation can also optionally implement `org.apache.isis.objectstore.jdo.datanucleus.DataNucleusPropertiesAware`:

```
public interface DataNucleusPropertiesAware {
    public void setDataNucleusProperties(final Map<String, String> properties);
}
```

This provides access to the properties passed through to JDO/DataNucleus.

If you do extend Apache Isis' `CreateSchemaObjectFromClassMetadata` class for some other database, please contribute back your improvements.

Chapter 5. Hints and Tips

This chapter provides some solutions for problems we've encountered ourselves or have been raised on the Apache Isis mailing lists.

See also hints-n-tips chapters in the:

- the Developers' guide
- the Wicket viewer guide
- the Restful Objects viewer guide
- the Datanucleus ObjectStore guide (this chapter)
- the Security guide
- the Beyond the Basics guide.

5.1. 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.

In fact, JDO/DataNucleus provides two different XML files that have slightly different purposes and capabilities:

- first, a `.jdo` file can be provided which - if found - completely replaces the annotations.

  The idea here is simply to use XML as the means by which metadata is specified.

- second, an `.orm` file can be provided which - if found - provides individual overrides for a particular database vendor.

  The idea here is to accommodate for subtle differences in support for SQL between vendors. A good example is the default schema for a table: `dbo` for SQL Server, `public` for HSQLDB, `sys` for Oracle, and so on.

If you want to use the first approach (the `.jdo` file), you'll find that you can download the effective XML representation of domain entities using the downloadJdoMetadata mixin action available in prototyping mode. This then needs to be renamed and placed in the appropriate location on the classpath; see the DataNucleus documentation for details.

However, using this first approach does create a maintenance effort; if the domain entity's class structure changes over time, then the XML metadata file will need to be updated.

The second approach (using an `.orm` file) is therefore often more useful than the first, because the metadata provided overrides rather than replaces the annotations (and annotations not overridden continue to be honoured).

A typical use case is to change the database schema for an entity. For example, the various (non-ASF) Incode Platform modules use schemas for each entity. For example, the `AuditEntry` entity in the (non-ASF) Incode Platform's 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. Also suppose we are using SQL Server as our target database.

We can override the above annotation using a `AuditEntry-sqlserver.orm` file, placed in the same package as the `AuditEntry` entity. For example:

AuditEntry-sqlserver.orm

```
<?xml version="1.0" encoding="UTF-8" ?>
<orm xmlns="http://xmlns.jcp.org/xml/ns/jdo/orm"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/orm
        http://xmlns.jcp.org/xml/ns/jdo/orm_3_0.xsd">

    <package name="org.isisaddons.module.audit.dom">
        <class name="AuditEntry"
               schema="isisaudit"
               table="AuditEntry">
        </class>
    </package>
</orm>
```

It's also necessary to tell JDO/DataNucleus about which vendor is being used (`sqlserver` in the example above). This is done using a configuration property:

isis.properties

```
isis.persistor.datanucleus.impl.datanucleus.Mapping=sqlserver
```

5.2. Subtype not fully populated

Taken from this thread on the Apache Isis users mailing list...

If it seems that Apache Isis (or rather DataNucleus) isn't fully populating domain entities (ie leaving some properties as `null`), then check that your actions are not accessing the fields directly. Use getters instead.

Properties of domain entities should always be accessed using getters. The only code that should access to fields should be the getters themselves.

Why so? Because DataNucleus will potentially lazy load some properties, but to do this it needs to know that the field is being requested. This is the purpose of the enhancement phase: the bytecode of the original getter method is actually wrapped in code that does the lazy loading checking. But hitting the field directly means that the lazy loading code does not run.

This error can be subtle: sometimes "incorrect" code that accesses the fields will seem to work. But that will be because the field has been populated already, for whatever reason.

One case where you will find the issue highlighted is for subtype tables that have been mapped using an inheritance strategy of `NEW_TABLE`, eg:

```
@javax.jdo.annotations.PersistenceCapable
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
public class SupertypeEntity {
    ...
}
```

and then:

```
@javax.jdo.annotations.PersistenceCapable
@javax.jdo.annotations.Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
public class SubtypeEntity extends SupertypeEntity {
    ...
}
```

This will generate two tables in the database, with the primary key of the supertype table propagated as a foreign key (also primary key) of the subtype table (sometimes called "table per type" strategy). This means that DataNucleus might retrieve data from only the supertype table, and the lazily load the subtype fields only as required. This is preferable to doing a left outer join from the super- to the subtype tables to retrieve data that might not be needed.

On the other hand, if the `SUPERCLASS_TABLE` strategy (aka "table per hierarchy" or roll-up) or the `SUBCLASS_TABLE` strategy (roll-down) was used, then the problem is less likely to occur because DataNucleus would obtain all the data for any given instance from a single table.

Final note: references to other objects (either scalar references or in collections) in particular require that getters rather than fields to be used to obtain them: it's hopefully obvious that DataNucleus (like all ORMs) should not and will not resolve such references (otherwise, where to stop… and the whole database would be loaded into memory).

In summary, there's just one rule: **always use the getters, never the fields**.

5.3. Java8

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

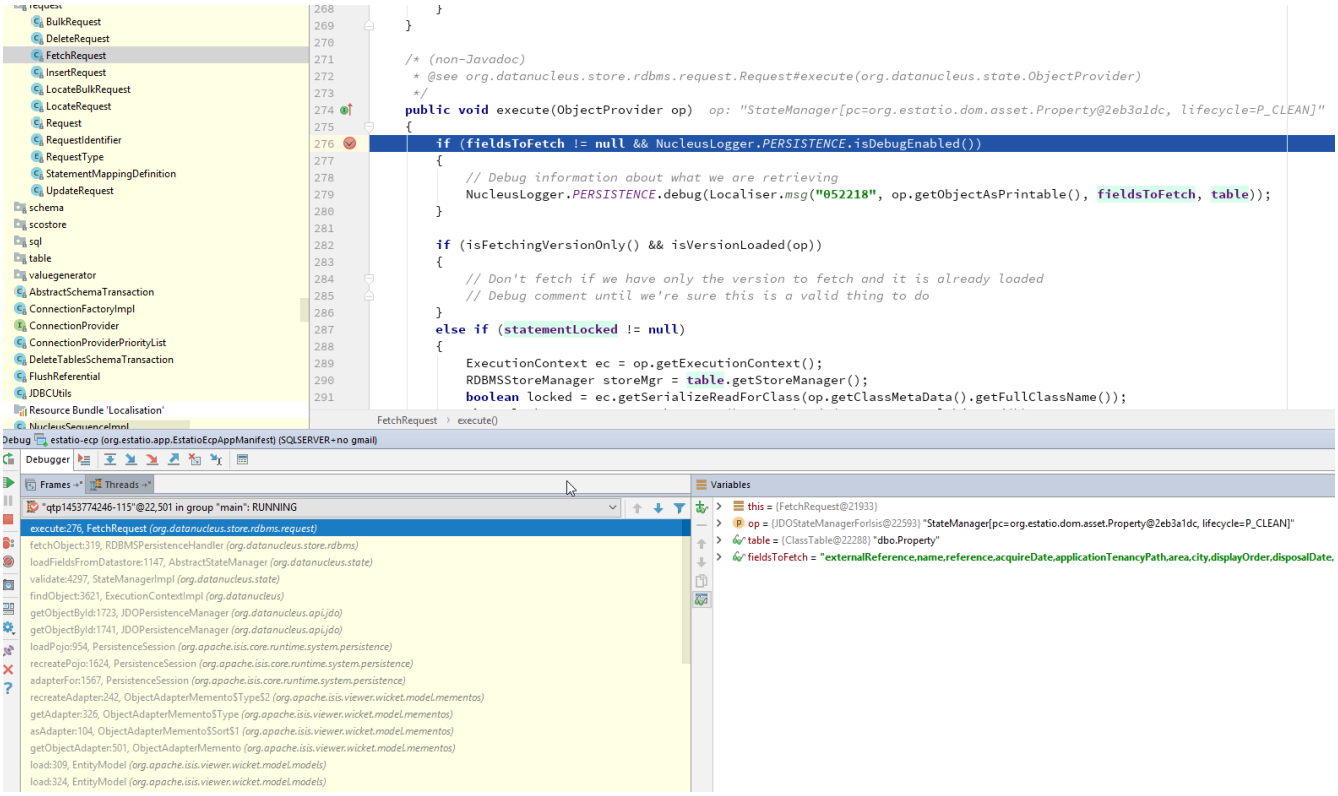
Just include within `<dependencies>` of your `dom` module's `pom.xml`:

```
<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.

5.4. Diagnosing n+1 Issues

(As of DN 4.1) set a break point in `FetchRequest#execute(…)`:



The "Variables" pane will tell you which field(s) are being loaded, and the stack trace should help explain why the field is required.

For example, it may be that an object is being loaded in a table and the initial query did not eagerly load that field. In such a case, consider using fetch groups in the initial repository query to bring the required data into memory with just one SQL call. See this hint/tip for further details.

5.5. Typesafe Queries and Fetch-groups

Fetch groups provide a means to hint to DataNucleus that it should perform a SQL join when querying. A common use case is to avoid the n+1 issue.

(So far as I could ascertain) it isn't possible to specify fetch group hints using JDOQL, but it is possible to specify them using the programmatic API or using typesafe queries.

For example, here's a JDOQL query:

```
@Query(
        name = "findCompletedOrLaterWithItemsByReportedDate", language = "JDOQL",
        value = "SELECT "
                + "FROM org.estatio.capex.dom.invoice.IncomingInvoice "
                + "WHERE items.contains(ii) "
                + "   && (ii.reportedDate == :reportedDate) "
                + "   && (approvalState != 'NEW' && approvalState != 'DISCARDED') "
                + "VARIABLES org.estatio.capex.dom.invoice.IncomingInvoiceItem ii "
),
public class IncomingInvoice ... { ... }
```

which normally would be used from a repository:

```
public List<IncomingInvoice> findCompletedOrLaterWithItemsByReportedDate(
        final LocalDate reportedDate) {
    return repositoryService.allMatches(
            new QueryDefault<>(
                    IncomingInvoice.class,
                    "findCompletedOrLaterWithItemsByReportedDate",
                    "reportedDate", reportedDate));
}
```

This can be re-written as a type-safe query as follows:

```
public List<IncomingInvoice> findCompletedOrLaterWithItemsByReportedDate(final
LocalDate reportedDate) {

    final QIncomingInvoice ii = QIncomingInvoice.candidate();
    final QIncomingInvoiceItem iii = QIncomingInvoiceItem.variable("iii");

    final TypesafeQuery<IncomingInvoice> q =
        isisJdoSupport.newTypesafeQuery(IncomingInvoice.class);

    q.filter(
            ii.items.contains(iii)
        .and(iii.reportedDate.eq(reportedDate))
        .and(ii.approvalState.ne(IncomingInvoiceApprovalState.NEW))
        .and(ii.approvalState.ne(IncomingInvoiceApprovalState.DISCARDED)));
    final List<IncomingInvoice> incomingInvoices = Lists.newArrayList(q.executeList()
);
    q.closeAll();
    return incomingInvoices;
}
```

Now the `IncomingInvoice` has four fields that require eager loading. This can be specified by defining a named fetch group:

```
@FetchGroup(
        name="seller_buyer_property_bankAccount",
        members={
                @Persistent(name="seller"),
                @Persistent(name="buyer"),
                @Persistent(name="property"),
                @Persistent(name="bankAccount")
        })
public class IncomingInvoice ... { ... }
```

This fetch group can then be used in the query using `q.getFetchPlan().addGroup(…)`. Putting this all together, we get:

```
public List<IncomingInvoice> findCompletedOrLaterWithItemsByReportedDate(final
LocalDate reportedDate) {

    final QIncomingInvoice ii = QIncomingInvoice.candidate();
    final QIncomingInvoiceItem iii = QIncomingInvoiceItem.variable("iii");

    final TypesafeQuery<IncomingInvoice> q =
        isisJdoSupport.newTypesafeQuery(IncomingInvoice.class);

    q.getFetchPlan().addGroup("seller_buyer_property_bankAccount");     ①

    q.filter(
            ii.items.contains(iii)
        .and(iii.reportedDate.eq(reportedDate))
        .and(ii.approvalState.ne(IncomingInvoiceApprovalState.NEW))
        .and(ii.approvalState.ne(IncomingInvoiceApprovalState.DISCARDED)));
    final List<IncomingInvoice> incomingInvoices = Lists.newArrayList(q.executeList()
);
    q.closeAll();
    return incomingInvoices;
}
```

① specify the fetch group to use.