DataNucleus ExtensionPoints Guide

Table of Contents

Chapter 1. Plugin Mechanism

DataNucleus products are built using a plugin mechanism, allowing plugins to operate together. This plugin mechanism involves the use of a file `plugin.xml` situated at the root of the CLASSPATH, containing a definition of the ExtensionPoints and Extensions that the plugin utilises/provides. The plugin mechanism originated in the Eclipse IDE, but has no dependencies on Eclipse. This plugin mechanism is useful also from a user viewpoint in that you, the user, could provide plugins that use these ExtensionPoints and extend the capabilities of DataNucleus. Plugins are loaded by a plugin manager when DataNucleus is initialised at runtime, and this plugin manager uses a registry mechanism, inspecting jars in the CLASSPATH.

A plugin loadable by DataNucleus requires 2 components.

1.1. plugin.xml

A file `plugin.xml` should be placed at the root of the plugin, and should be something like this (refer to the specific ExtensionPoint(s) being provided for precise content).

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.type_converter">
    <type-converter name="dn.uri-string" member-type="java.net.URI" datastore-
type="java.lang.String" converter-class="mydomain.converters.URIStringConverter"/>
    </extension>
</plugin>
```

1.2. MANIFEST.MF

A minimum `META-INF/MANIFEST.MF` for a plugin jar should look like this

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: myplugin
Bundle-SymbolicName: mydomain.myplugin
Bundle-Version: 1.0.0
Bundle-Vendor: My Company
```

This provides basic information for DataNucleus, but also for the OSGi environment, should it ever be required.

Each DataNucleus bundle uses a version schema that has the following versioning scheme: *major.minor.revision.qualifier* major, minor and revision are numeric values, and qualifier is an alphanumeric.

Each bundle has the version value set in the `/META-INF/MANIFEST.MF` file, Bundle-Version entry.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: DataNucleus Enhancer
Bundle-SymbolicName: org.datanucleus.enhancer;singleton:=true
Bundle-Vendor: DataNucleus
Bundle-Version: 1.2.0.b2
```

The most common version compatibility policies are:

- major - An incompatible update
- minor - A backward compatible update
- revision - A change that does not affect the interface: for example, a bug fix

When your bundle depends on another bundle, you must declare it in the MANIFEST file, via the Require-Bundle entry.

For example, the RDBMS plugin (org.datanucleus.store.rdbms) plug-in depends on the core (org.datanucleus) plug-in.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: DataNucleus RDBMS
Bundle-SymbolicName: org.datanucleus.store.rdbms;singleton:=true
Bundle-Vendor: DataNucleus
Bundle-Version: 1.2.0.b2
Bundle-Localization: plugin
Require-Bundle: org.datanucleus
```

See more in the OSGi 3.0 specification §3.13.1.

If a bundle depends on a specific version of a bundle, you must declare it in the *Require-Bundle* entry, bundle-version parameter. For example

```
Require-Bundle: org.datanucleus;bundle-version=(1.2.0.b2, 2.0)
```

See more in the OSGi 3.0 specification §3.2.5 and §3.13.1 chapters.

All DataNucleus plugins use the Maven *bundle* plugin to auto-generate the `MANIFEST.MF` file. This means that the version in the `pom.xml` is taken for the bundle version, and the dependencies are auto-generated from imports etc. We recommend that you use this same method for your own plugins.

1.3. Plugins in a Non-managed environment

A non-managed environment is a runtime environment where DataNucleus runs and plug-ins are not managed by a container. In this environment the plug-in discovery and lifecycle is managed by

DataNucleus. JavaSE and JavaEE runtimes are considered *non-managed* environments. In non managed environments there is no lifecycle itself for plug-ins.

There is a 1 to N instance relationship from DataNucleus to a plug-in per PMF/EMF. More exactly, if only one PMF/EMF exists, there is only one Plug-in instance for a Connection Pool Plug-in, and if "N" PMF/EMF exist, there are "N" Plug-in instances for a ConnectionPool Plug-in. Extensions implemented by plugins are instantiated on demand and terminated on PMF/EMF closing, PM/EM closing or in another form depending on what the extension is used for.

1.4. Plugins in a Managed environment

Managed environment is a runtime environment where DataNucleus plug-ins are managed by a container. The discovery, registry and lifecycle of plug-ins are controlled by the container. There is no plug-in instance relationship from DataNucleus to a plug-in regarding PMF/EMF instances. In managed environments, there is only one plug-in instance for one or "N" PMF/EMFs. Again, this is managed by the container.

DataNucleus supports OSGi containers as a managed environment. In OSGi managed environments the plug-in lifecycle is determined by the OSGi specification. Once activated, a plug-in is only stopped when the OSGi container finishes its execution, or the plug-in is stopped by an OSGi command.

1.5. Extensions and Plugins

A plugin owns an Extension, and an Extension implements an ExtensionPoint. The behaviour is defined below :-

- *Lifecycle* : Each extension is created by a segment of code during the runtime execution, and destroyed/released whenever they are no longer needed. This has no influence with the plug-in lifecycle.
- *Manageability* : In non managed environments, the plug-ins are managed by DataNucleus and maintained with a composition relation to the PMF/EMF instance. This allows a plug-in "instance" per PMF/EMF. If multiple PMF/EMFs are created multiple extensions for an ExtensionPoint are instantiated. In OSGi managed environments, the plug-ins are managed by the OSGi framework, and each plug-in will mostly be a singleton inside the OSGi container.
- *Registration* : In non managed environments all plugins are registered using an instance of JDOClassLoaderResolver (so using the current ClassLoader of the PMF/EMF and the current thread). This means that the `/plugin.xml` and `/META-INF/MANIFEST.MF` files must be accessible to the classloader. In managed environment this is handled by the container.
- *ClassLoading* : The classloading in non managed environments is usually made of one single ClassLoader, while in managed environments each plug-in has it’s own ClassLoader.
- *Configuration* : Some Extensions needs to retrieve a configuration that was set in the PMF/EMF. This means that Plug-ins should not hold singleton / static configurations if they want to serve to multiple PMFs at the same time.
- *Constructors/Methods* : In order of having consistent and avoid changes to ExtensionPoint interfaces, the Extension Constructors or Methods (either one) should have receive a

NucleusContext instance as argument. If by the time the ExtensionPoint is designed clearly there is usage for a NucleusContext, then the ExtensionPoint does not need to take the NucleusContext as argument, but keep in mind that a 3rd Extension may need one due to different reasons.

- *Instantiation* : Inside dataNucleus-core, regardless if the runtime is OSGi managed or non managed, extension instances are created per PMF/EMF. DataNucleus Extensions should always be created through a PluginManager, regardless if the managed environment would allow you to instantiate using their own interfaces. This allows DataNucleus and its Plug-ins to run in non managed environments.

Each ExtensionPoint has attributes. If you want to override an extension that is included in DataNucleus itself then you need to specify the priority attribute, setting it to an integer (the default plugin has priority=0, so set to higher than this to override it).

If you write a DataNucleus plugin and you either want it to be included in the DataNucleus distribution, or want it to be listed here then please contact us.

Chapter 2. Mapping ExtensionPoints

2.1. Annotations

Certain annotations are used to detect whether a class is involved in the persistence process; this is extensible. DataNucleus provides support for JDO and JPA annotations, but is structured so that you can easily add your own annotations and have them usable within your DataNucleus usage.

The `datanucleus-api-jdo` plugin provides support for JDO annotations, and the `datanucleus-api-jpa` plugin provides support for JPA annotations. You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.annotations**.

Plugin extension-point	Key	Description	Location
org.datanucleus.annotations	@PersistenceCapable	JDO annotation reader	datanucleus-api-jdo
org.datanucleus.annotations	@PersistenceAware	JDO annotation reader	datanucleus-api-jdo
org.datanucleus.annotations	@Entity	JPA annotation reader	datanucleus-api-jpa
org.datanucleus.annotations	@MappedSuperclass	JPA annotation reader	datanucleus-api-jpa
org.datanucleus.annotations	@Embeddable	JPA annotation reader	datanucleus-api-jpa

2.1.1. Interface

Any annotation reader plugin will need to implement *org.datanucleus.metadata.annotations.AnnotationReader*. Javadoc. So you need to implement the following interface

```
package org.datanucleus.metadata.annotations;

import org.datanucleus.metadata.PackageMetaData;
import org.datanucleus.metadata.ClassMetaData;

public interface AnnotationReader
{
    /**
     * Accessor for the annotations packages supported by this reader.
     * @return The annotations packages that will be processed.
     */
    String[] getSupportedAnnotationPackages();

    /**
     * Method to get the ClassMetaData for a class from its annotations.
     * @param cls The class
     * @param pmd MetaData for the owning package (that this will be a child of)
     * @return The ClassMetaData (unpopulated and unitialised)
     */
    public ClassMetaData getMetaDataForClass(Class cls, PackageMetaData pmd);
}
```

2.1.2. Plugin Specification

So we now have our custom "annotation reader" and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. This file should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.annotations" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.annotations">
        <annotations annotation-class="mydomain.annotations.MyAnnotationType"
reader="mydomain.annotations.MyAnnotationReader"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

So here we have our "annotations reader" class "MyAnnotationReader" which will process any classes annotated with the annotation "MyAnnotationType".

2.2. ClassAnnotationHandler

DataNucleus supports some annotations defined at **class** level. The supported annotations at this level is extensible, and so you can easily add your own annotations and have them usable within your DataNucleus usage.

2.2.1. Interface

Any class annotation handler plugin will need to implement *org.datanucleus.metadata.annotations.ClassAnnotationHandler* Javadoc. So you need to implement the following interface

```
package org.datanucleus.metadata.annotations;

import org.datanucleus.ClassLoaderResolver;
import org.datanucleus.metadata.AbstractClassMetaData;

public interface ClassAnnotationHandler
{
    /**
     * Method to process a class level annotation.
     * @param annotation The annotation
     * @param cmd Metadata for the class to update with any necessary information.
     * @param clr ClassLoader resolver
     */
    void processClassAnnotation(AnnotationObject annotation, AbstractClassMetaData
cmd, ClassLoaderResolver clr);
}
```

2.2.2. Plugin Specification

So we now have our custom "annotation handler" and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. This file should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.annotations" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.class_annotation_handler">
        <class-annotation-handler annotation-class="mydomain.annotations.MyAnnotation"
handler="mydomain.annotations.MyAnnotationHandler"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

So here, when the metadata for our class is processed, if it finds the @MyAnnotation annotation it will call this handler after generating the basic metadata for the class, allowing us to update it.

2.3. MemberAnnotationHandler

DataNucleus supports some annotations defined at **member** (field/getter) level. The supported annotations at this level is extensible, and so you can easily add your own annotations and have

them usable within your DataNucleus usage.

2.3.1. Interface

Any member annotation handler plugin will need to implement *org.datanucleus.metadata.annotations.MemberAnnotationHandler* Javadoc. So you need to implement the following interface

```
package org.datanucleus.metadata.annotations;

import org.datanucleus.ClassLoaderResolver;
import org.datanucleus.metadata.AbstractMemberMetaData;

public interface MemberAnnotationHandler
{
    /**
     * Method to process a member level annotation.
     * @param annotation The annotation
     * @param mmd Metadata for the member to update with any necessary information.
     * @param clr ClassLoader resolver
     */
    void processMemberAnnotation(AnnotationObject annotation, AbstractMemberMetaData
cmd, ClassLoaderResolver clr);
}
```

2.3.2. Plugin Specification

So we now have our custom "annotation handler" and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. This file should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.annotations" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.member_annotation_handler">
        <member-annotation-handler annotation-
class="mydomain.annotations.MyAnnotation"
                handler="mydomain.annotations.MyAnnotationHandler"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

So here, when the metadata for a member is processed, if it finds the @MyAnnotation annotation it will call this handler after generating the basic metadata for the member, allowing us to update it.

2.4. XML Metadata Handler

DataNucleus has supported XML metadata from the outset. More than this, it actually provides a pluggable framework whereby you can plug in your own XML MetaData support. DataNucleus provides JDO XML support (in `datanucleus-api-jdo`), JPA XML support (in `datanucleus-api-jpa`), as well as `persistence.xml` (in `datanucleus-core`). You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.metadata_handler**.

2.5. Metadata EntityResolver

DataNucleus allows you to add in your own DTD or XSD schema support. You can extend DataNucleus's capabilities using the plugin extension *org.datanucleus.metadata_entityresolver*.

An Entity Resolver extension point configuration is a list of XSDs schemas or DTDs. For XSD schemas, it needs only the schema location (URL in classpath), since the XML parser will take care of association of XML elements and namespaces to the schemas types. For DTDs, it's a little less obvious since DTDs are not namespace aware, so the configuration must contain the URL location (location in classpath), DTD DOCTYPE (PUBLIC or SYSTEM), and the DTD identity.

An Entity Resolver has close relationship to the MetaData Handler extension point, since the MetaData Handler uses the Entity Resolvers to validate XML files. However, the Entity Resolvers configured via this plugin are only applicable to the *org.datanucleus.metadata.xml.PluginEntityResolver* instances, so the MetaData Handler must use the *PluginEntityResolver* Entity Resolver if it wants to use the schemas configured in this extension point.

2.5.1. Implementation

The Entity Resolver implementation must extend the *org.datanucleus.util.AbstractXMLEntityResolver* class and must have a public constructor that takes the *org.datanucleus.plugin.PluginManager* class.

```
import org.datanucleus.plugin.PluginManager;
import org.datanucleus.util.AbstractXMLEntityResolver;

/**
 * Implementation of an entity resolver for DTD/XSD files.
 * Handles entity resolution for files configured via plugins.
 */
public class PluginEntityResolver extends AbstractXMLEntityResolver
{
    public PluginEntityResolver(PluginManager pm)
    {
        publicIdEntities.put("identity...", "url....");
        ...
        systemIdEntities.put("identity...", "url....");
        ...
    }
}
```

Chapter 3. Cache ExtensionPoints

3.1. Level 1 Cache

DataNucleus comes with built-in support for three Level 1 caches (weak, soft and strong references), but this is also an extension point so that you can easily add your own variant and have it usable within your DataNucleus usage.

You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.cache_level1**.

The following sections describe how to create your own Level 1 cache plugin for DataNucleus.

3.1.1. Interface

If you have your own Level1 cache you can easily use it with DataNucleus. DataNucleus defines a Level1Cache interface and you need to implement this. Javadoc.

```
package org.datanucleus.cache;

public interface Level1Cache extends Map
{
}
```

So you need to create a class, *MyLevel1Cache* for example, that implements this interface (i.e that implements *java.util.Map*).

3.1.2. Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file `plugin.xml` in your JAR at the root of the CLASSPATH, like this

```
<?xml version="1.0"?>
<plugin id="mydomain.mycache" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.cache_level1">
        <cache name="MyCache" class-name="mydomain.MyLevel1Cache"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

3.1.3. Plugin Usage

The only thing remaining is to use your L1 Cache plugin. To do this you specify the persistence property **datanucleus.cache.level1.type** as *MyCache* (the "name" in `plugin.xml`).

3.2. Level 2 Cache

DataNucleus provides a large selection of Level 2 caches (built-in map-based, soft, weak, javax.cache, Coherence, EHCache, OSCache, others) but is structured so that you can easily add your own variant and have it usable within your DataNucleus usage.

You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.cache_level2**.

The following sections describe how to create your own Level 2 cache plugin for DataNucleus.

3.2.1. Interface

If you have your own Level2 cache you can easily use it with DataNucleus. DataNucleus defines a Level2Cache interface and you need to implement this. Javadoc.

```
package org.datanucleus.cache;
public interface Level2Cache
{
    void close();

    void evict (Object oid);
    void evictAll ();
    void evictAll (Object[] oids);
    void evictAll (Collection oids);
    void evictAll (Class pcClass, boolean subclasses);

    void pin (Object oid);
    void pinAll (Collection oids);
    void pinAll (Object[] oids);
    void pinAll (Class pcClass, boolean subclasses);

    void unpin(Object oid);
    void unpinAll(Collection oids);
    void unpinAll(Object[] oids);
    void unpinAll(Class pcClass, boolean subclasses);

    int getNumberOfPinnedObjects();
    int getNumberOfUnpinnedObjects();
    int getSize();
    CachedPC get(Object oid);
    CachedPC put(Object oid, CachedPC pc);
    boolean isEmpty();
    void clear();
    boolean containsOid(Object oid);
}
```

3.2.2. Implementation

Let's suppose your want to implement your own Level 2 cache *MyLevel2Cache*

```
package mydomain;

import org.datanucleus.NucleusContext;
import org.datanucleus.cache.Level2Cache;

public class MyLevel2Cache implements Level2Cache
{
    /**
     * Constructor.
     * @param nucCtx Nucleus Context
     */
    public MyLevel2Cache(NucleusContext nucCtx)
    {
        ...
    }

    ... (implement the interface)
}
```

3.2.3. Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file `plugin.xml` in your JAR at the root of the CLASSPATH, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.cache_level2">
        <cache name="MyCache" class-name="mydomain.MyLevel2Cache"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

3.2.4. Plugin Usage

The only thing remaining is to use your L2 Cache plugin. To do this you specify the persistence property **datanucleus.cache.level2.type** as *MyCache* (the "name" in `plugin.xml`).

3.3. Query Cache

When a query is created in DataNucleus it will typically be compiled, and this compilation can be cached to save re-compilation of the same query later on. DataNucleus provides some inbuilt cache options (soft/weak/strong references, javax.cache, EHCache etc) but also allows you to provide your

own.

You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.cache_query**.

The following sections describe how to create your own Query cache plugin for DataNucleus.

3.3.1. Interface

If you have your own Query cache you can easily use it with DataNucleus. DataNucleus defines a QueryCompilationCache interface and you need to implement this. Javadoc.

```
package org.datanucleus.store.query.cache;
public interface QueryCompilationCache
{
    void close();
    void evict(String queryKey);
    void clear();
    boolean isEmpty();
    int size();
    QueryCompilation get(String queryKey);
    QueryCompilation put(String queryKey, QueryCompilation compilation);
    boolean contains(String queryKey);
}
```

3.3.2. Implementation

Let's suppose your want to implement your own Level 2 cache *MyLevel2Cache*

```
package mydomain;

import org.datanucleus.NucleusContext;
import org.datanucleus.store.query.compiler.QueryCompilationCache;

public class MyQueryCache implements QueryCompilationCache
{
    public MyQueryCache(NucleusContext nucCtx)
    {
        ...
    }

    ... (implement the interface)
}
```

3.3.3. Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file `plugin.xml` in your JAR at the root of the CLASSPATH, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.cache_query">
        <cache name="MyCache" class-name="mydomain.MyQueryCache"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

3.3.4. Plugin Usage

The only thing remaining is to use your Query Compilation Cache plugin. To do this you specify the persistence property **datanucleus.cache.query.type** as *MyCache* (the "name" in `plugin.xml`).

Chapter 4. Identity ExtensionPoints

4.1. DatastoreIdentity

When a class is defined to use "datastore" (surrogate) identity, it will internally use a particular class to represent that identity. DataNucleus provides its own built-in default (*org.datanucleus.identity.DatastoreIdImpl*) that generates ids using the String form like "3286[OID]mydomain.MyClass", but this feature is extensible Having this component configurable means that you can override the output of the toString() to be more suitable for any use of these identities. Please be aware that the JDO specification (5.4.3) has strict rules for datastore identity classes.

You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.store_datastoreidentity**. It provides the following internally

Plugin extension-point	Key	Description	Location
org.datanucleus.store_datastoreidentity	datanucleus	Datastore Identity used by DataNucleus since DataNucleus 1.0 ("1[OID]org.datanucleus.myClass")	datanucleus-core
org.datanucleus.store_datastoreidentity	kodo	Datastore Identity in the style of OpenJPA/Kodo ("org.datanucleus.myClass-1")	datanucleus-core
org.datanucleus.store_datastoreidentity	xcalia	Datastore Identity in the style of Xcalia ("org.datanucleus.myClass:1"). This ignores Xcalias support for class aliases	datanucleus-core

The following sections describe how to create your own datastore identity plugin for DataNucleus.

4.1.1. Interface

Any datastore identity plugin will need to implement *org.datanucleus.identity.DatastoreId* Javadoc So you need to implement the following interface

```
import org.datanucleus.identity;

public interface DatastoreId
{
    /**
     * Provides the datastore id in a form that can be used by the database as a key.
     * @return The key value
     */
    public Object getKeyAsObject();

    /**
     * Accessor for the PC class name
     * @return the PC Class
     */
    public String getTargetClassName();

    /**
     * Equality operator.
     * @param obj Object to compare against
     * @return Whether they are equal
     */
    public abstract boolean equals(Object obj);

    /**
     * Accessor for the hashcode
     * @return Hashcode for this object
     */
    public abstract int hashCode();

    /**
     * Returns the string representation of the datastore id.
     * The string representation should contain enough information to be usable as
input to a String constructor
     * to create the datastore id.
     * @return the string representation of the datastore id.
     */
    public abstract String toString();
}
```

4.1.2. Implementation

DataNucleus provides an abstract base class *org.datanucleus.identity.DatastoreIdImpl* as a guideline. The DataNucleus internal implementation is defined as

```
package org.datanucleus.identity;

public class DatastoreIdImpl implements java.io.Serializable, DatastoreId
{
    /** Separator to use between fields. */
```

```
    private transient static final String STRING_DELIMITER = "[OID]";

    // JDO spec 5.4.3 - serializable fields required to be public.

    public final Object keyAsObject;
    public final String targetClassName;

    /** pre-created toString to improve performance **/
    public final String toString;

    /** pre-created hasCode to improve performance **/
    public final int hashCode;

    public DatastoreIdImpl()
    {
        keyAsObject = null;
        targetClassName = null;
        toString = null;
        hashCode = -1;
    }

    public DatastoreIdImpl(String pcClass, Object object)
    {
        this.targetClassName = pcClass;
        this.keyAsObject = object;

        StringBuilder s = new StringBuilder();
        s.append(this.keyAsObject.toString());
        s.append(STRING_DELIMITER);
        s.append(this.targetClassName);
        toString = s.toString();
        hashCode = toString.hashCode();
    }

    /**
     * Constructs an identity from its string representation that is consistent with
the output of toString().
     * @param str the string representation of a datastore id
     * @exception IllegalArgumentException if the given string representation is not
valid.
     * @see #toString
     */
    public DatastoreIdImpl(String str)
    throws IllegalArgumentException
    {
        if (str.length() < 2)
        {
            throw new IllegalArgumentException(Localiser.msg("038000", str));
        }
        else if (str.indexOf(STRING_DELIMITER) < 0)
        {
```

```
            throw new IllegalArgumentException(Localiser.msg("038000", str));
        }

        int start = 0;
        int end = str.indexOf(STRING_DELIMITER, start);
        String oidStr = str.substring(start, end);
        Object oidValue = null;
        try
        {
            // Use Long if possible, else String
            oidValue = Long.valueOf(oidStr);
        }
        catch (NumberFormatException nfe)
        {
            oidValue = oidStr;
        }
        keyAsObject = oidValue;

        start = end + STRING_DELIMITER.length();
        this.targetClassName = str.substring(start, str.length());

        toString = str;
        hashCode = toString.hashCode();
    }

    public Object getKeyAsObject()
    {
        return keyAsObject;
    }

    public String getTargetClassName()
    {
        return targetClassName;
    }

    public boolean equals(Object obj)
    {
        if (obj == null)
        {
            return false;
        }
        if (obj == this)
        {
            return true;
        }
        if (!(obj.getClass().getName().equals(ClassNameConstants.IDENTITY_OID_IMPL)))
        {
            return false;
        }
        if (hashCode() != obj.hashCode())
        {
```

```
            return false;
        }
        if (!((DatastoreId)obj).toString().equals(toString))
        {
            // Hashcodes are the same but the values aren't
            return false;
        }
        return true;
    }

    public int compareTo(Object o)
    {
        if (o instanceof DatastoreIdImpl)
        {
            DatastoreIdImpl c = (DatastoreIdImpl)o;
            return this.toString.compareTo(c.toString);
        }
        else if (o == null)
        {
            throw new ClassCastException("object is null");
        }
        throw new ClassCastException(this.getClass().getName() + " != " + o.
getClass().getName());
    }

    public int hashCode()
    {
        return hashCode;
    }

    /**
     * Creates a String representation of the datastore identity, formed from the
target class name and the key value. This will be something like
     * <pre>3254[OID]mydomain.MyClass</pre>
     * @return The String form of the identity
     */
    public String toString()
    {
        return toString;
    }
}
```

As show you need 3 constructors. One is the default constructor. One takes a String (which is the output of the toString() method). The other takes the PC class name and the key value.

4.1.3. Plugin Specification

So once we have our custom "datastore identity" we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. This file should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_datastoreidentity">
        <datastoreidentity name="myoid" class-name="mydomain.MyOIDImpl" unique=
"true"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

You should now create the PMF/EMF using the persistence property **datanucleus.datastoreIdentityType** set to *myoid*. Thats all. You now have a DataNucleus "datastore identity" extension.

4.2. IdentityKeyTranslator

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is translation of identities. When you call *pm.getObjectById(cls, key)* you pass in the "key". This object can be the toString() form of an identity, or the key of a single-string form. Some store managers (e.g GAE/J) allow non-standard key input and this allows for the translation into a standardised key form. Alternatively you could do this in your own code, but the facility is provided. This means that in your application you only use your own form of identities.

You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.identity_key_translator**.

4.2.1. Interface

Any identifier factory plugin will need to implement *org.datanucleus.store.IdentifierKeyFactory*. Javadoc. So you need to implement the following interface

```
package org.datanucleus.identity;

public interface IdentityKeyTranslator
{
    /**
     * Method to translate the string into the identity.
     * @param ec ExecutionContext
     * @param cls The persistable class
     * @param key The input key
     * @return The returned key
     */
    Object getKey(ObjectManager om, Class cls, Object key);
}
```

4.2.2. Plugin Specification

When we have defined our "IdentityKeyTranslator" we just need to make it into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. This file should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.identity_key_translator">
        <identitykeytranslator name="mytranslator" class-
name="mydomain.MyIdKeyTranslator"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

4.2.3. Plugin Usage

The only thing remaining is to use your new *IdentityStringTranslator* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the persistence property **datanucleus.identityKeyTranslatorType** to *mytranslator* (the name you specified in the `plugin.xml` file).

4.3. IdentityStringTranslator

When you call *pm.getObjectById(id)* you pass in an object. This object can be the toString() form of an identity. Some other JDO implementations (e.g Xcalia) allowed non-standard String input here including a discriminator. This extension point allows for such non-standard String input forms to pm.getObjectById(id) and can provide a plugin that translates this String into a valid JDO identity. Alternatively you could do this in your own code, but the facility is provided. This means that in your application you only use your own form of identities.

You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.identity_string_translator**.

Plugin extension-point	Key	Description	Location
org.datanucleus.identity_string_translator	xcalia	Translator that allows for {discriminator}:key as well as the usual input, as supported by Xcalia XIC	datanucleus-core

4.3.1. Interface

Any identifier factory plugin will need to implement *org.datanucleus.store.IdentifierStringFactory*.

Javadoc. So you need to implement the following interface

```
package org.datanucleus.identity;

public interface IdentityStringTranslator
{
    /**
     * Method to translate the string into the identity.
     * @param om ObjectManager
     * @param stringId String form of the identity
     * @return The identity
     */
    Object getIdentity(ObjectManager om, String stringId);
}
```

4.3.2. Plugin Specification

When we have defined our "IdentityStringTranslator" we just need to make it into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. This file should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.identity_string_translator">
        <identitystringtranslator name="mytranslator" class-
name="mydomain.MyIdStringTranslator"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

4.3.3. Plugin Usage

The only thing remaining is to use your new *IdentityStringTranslator* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the persistence property **datanucleus.identityStringTranslatorType** to *mytranslator* (the name you specified in the `plugin.xml` file).

Chapter 5. Type ExtensionPoints

5.1. Java Types

DataNucleus provides capabilities for persistence of particular Java types. Some types are by default persistent, some are by default in the default "fetch-group". Similarly some are second class mutable, and hence have their operations intercepted. An *extension-point* is available to define other Java types in this way. You can extend DataNucleus's capabilities to support a particular Java type using the plugin extension **org.datanucleus.java_type**.

The attributes that you can set for each Java type are

- **dfg** - whether this type is by default in the default-fetch-group (true/false)
- **embedded** - whether this type is, by default, embedded (true/false)
- **wrapper-type** - class name of the SCO wrapper (if it needs a wrapper)
- **wrapper-type-backed** - class name of a SCO wrapper (with backing store)
- **converter-name** - name of a TypeConverter to use as the *default* way of persisting this type when it isn't directly persistable as-is. Please refer to Type Converter extension point for details of how to define your own type converter
- **container-handler** - name of a ContainerHandler. Required for SCO Container types in DN. SCO Containers can contain other FCOs or SCOs objects e.g.: collections, maps and arrays.
- **priority** - Set this to a large number if you are overriding the default handling for a type that is supported out of the box

All of these are optional, and you should define what is required for your type.

5.1.1. wrapper-type

As we've mentioned above, if a java type is considered *second class mutable* then it needs to have any mutating operations intercepted. The reason for this is that DataNucleus needs to be aware when the type has changed value internally. To give an example of such a type and how you would define support for intercepting these mutating operations lets use *java.util.Date*. We need to write a *wrapper* class. This has to be castable to the same type as the Java type it is representing (so inherited from it). So we extend "java.util.Date", and we need to implement the interface *org.datanucleus.store.types.SCO* Javadoc.

```
package org.mydomain;

import java.io.ObjectStreamException;
import javax.jdo.JDOHelper;
import javax.jdo.spi.PersistenceCapable;
import org.datanucleus.state.ObjectProvider;

public class MyDateWrapper extends java.util.Date implements SCO
{
```

```
    private transient ObjectProvider ownerOP;
    private transient String fieldName;

    public MyDateWrapper(ObjectProvider op, String fieldName)
    {
        super();

        this.ownerOP = op;
        this.fieldName = fieldName;
    }

    public void initialise()
    {
    }

    /** Method to initialise the SCO from an existing value. */
    public void initialise(Object o, boolean forInsert, boolean forUpdate)
    {
        super.setTime(((java.util.Date)o).getTime());
    }

    /** Wrapper for the setTime() method. Mark the object as "dirty" */
    public void setTime(long time)
    {
        super.setTime(time);
        makeDirty();
    }

    /** Wrapper for the setYear() deprecated method. Mark the object as "dirty" */
    public void setYear(int year)
    {
        super.setYear(year);
        makeDirty();
    }

    /** Wrapper for the setMonth() deprecated method. Mark the object as "dirty" */
    public void setMonth(int month)
    {
        super.setMonth(month);
        makeDirty();
    }

    /** Wrapper for the setDates() deprecated method. Mark the object as "dirty" */
    public void setDate(int date)
    {
        super.setDate(date);
        makeDirty();
    }

    /** Wrapper for the setHours() deprecated method. Mark the object as "dirty" */
    public void setHours(int hours)
```

```
    {
        super.setHours(hours);
        makeDirty();
    }

    /** Wrapper for the setMinutes() deprecated method. Mark the object as "dirty" */
    public void setMinutes(int minutes)
    {
        super.setMinutes(minutes);
        makeDirty();
    }

    /** Wrapper for the setSeconds() deprecated method. Mark the object as "dirty" */
    public void setSeconds(int seconds)
    {
        super.setSeconds(seconds);
        makeDirty();
    }

    /** Accessor for the unwrapped value that we are wrapping. */
    public Object getValue()
    {
        return new java.util.Date(getTime());
    }

    public Object clone()
    {
        Object obj = super.clone();
        ((Date)obj).unsetOwner();
        return obj;
    }

    public void unsetOwner()
    {
        ownerOP = null;
    }

    public Object getOwner()
    {
        return (ownerOP != null ? ownerOP.getObject() : null);
    }

    public String getFieldName()
    {
        return this.fieldName;
    }

    public void makeDirty()
    {
        if (ownerSM != null)
        {
```

```
            ownerSM.getObjectManager().getApiAdapter().makeFieldDirty(owner,
fieldName);
        }
    }

    public Object detachCopy(FetchPlanState state)
    {
        return new java.util.Date(getTime());
    }

    public void attachCopy(Object value)
    {
        long oldValue = getTime();
        initialise(value, false, true);

        // Check if the field has changed, and set the owner field as dirty if
necessary
        long newValue = ((java.util.Date)value).getTime();
        if (oldValue != newValue)
        {
            makeDirty();
        }
    }

    /**
     * Handling for serialising our object.
     */
    protected Object writeReplace() throws ObjectStreamException
    {
        return new java.util.Date(this.getTime());
    }
}
```

So we simply intercept the mutators and mark the object as dirty in its StateManager.

5.1.2. Plugin Specification

To define the persistence characteristics of a Java type you need to add entries to a `plugin.xml` file at the root of the CLASSPATH. The file `plugin.xml` will look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.mystore" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.java_type">
        <java-type name="java.util.Date" wrapper-type="mydomain.MyDateWrapper"
dfg="true" priority="10"/>
    </extension>
</plugin>
```

Note that the *priority* is specified since this type is provided by DataNucleus itself and so your

mapping needs to override it. Note also that you require a `MANIFEST.MF` file as described above.

Obviously all standard types (such as *java.util.Date*) already have their values defined by DataNucleus itself typically in *datanucleus-core*.

5.2. Type Converter

DataNucleus allows you to provide alternate ways of persisting Java types. Whilst it includes the majority of normal converters built-in, you can extend DataNucleus capabilities using the plugin extension **org.datanucleus.type_converter**.

5.2.1. TypeConverter Interface

Any type converter plugin will need to implement *org.datanucleus.store.types.converters.TypeConverter* Javadoc. So you need to implement the following interface

```
public interface TypeConverter<X, Y> extends Serializable
{
    /**
     * Method to convert the passed member value to the datastore type.
     * @param memberValue Value from the member
     * @return Value for the datastore
     */
    Y toDatastoreType(X memberValue);

    /**
     * Method to convert the passed datastore value to the member type.
     * @param datastoreValue Value from the datastore
     * @return Value for the member
     */
    X toMemberType(Y datastoreValue);
}
```

5.2.2. TypeConverter Implementation Example

Let's take an example. If we look at the Java type URI we want to persist it as a String since a native URI type isn't present in datastores. We define our class as

```
public class URIStringConverter implements TypeConverter<URI, String>
{
    public URI toMemberType(String str)
    {
        if (str == null)
        {
            return null;
        }

        return java.net.URI.create(str.trim());
    }

    public String toDatastoreType(URI uri)
    {
        return uri != null ? uri.toString() : null;
    }
}
```

So when converting it for the datastore it will use the *toString()* form of the URI, and will be converted back to a URI (on retrieval from the datastore) using the *URI.create* method. Obviously this particular TypeConverter is included in DataNucleus, but hopefully it gives an idea of what to do to provide your own.

5.2.3. Controlling default column length

Some datastore plugins may support schemas where you can put an upper limit on the length of columns (e.g RDBMS). You can build this information into your TypeConverter plugin by also implementing the interface ColumnLengthDefiningTypeConverter Javadoc which simply means adding the method *int getDefaultColumnLength(int columnPosition)*.

5.2.4. Converting a member to multiple columns

The default is to convert a member type to a single column type in the datastore. DataNucleus allows you to convert to multiple columns, for example imagine a type Point that has an *x* and *y*. You want to persist this into 2 columns, the *x* stored in column 0, and the *y* stored in column 1. So now you update your TypeConverter to also implement MultiColumnConverter Javadoc which simply means adding the method *Class[] getDatastoreColumnTypes()*.

5.2.5. Plugin Specification

So we now have our custom "value generator" and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. The file `plugin.xml` should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.type_converter">
        <type-converter name="dn.uri-string" member-type="java.net.URI" datastore-
type="java.lang.String"
            converter-class="mydomain.converters.URIStringConverter"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

The name "dn.uri-string" can be used to refer to this converter from within a java_types extension point definition for the default converter to use for a Java type.

Chapter 6. Other ExtensionPoints

6.1. Persistence Property

DataNucleus persistence is controlled by persistence properties, specified at runtime. These properties are used by plugins and allow configurability of behaviour. Any plugin can define its own properties.

You can define persistence properties using the plugin extension **org.datanucleus.persistence_properties**.

6.1.1. Plugin Specification

The only thing required is to register the persistence property with DataNucleus via the `plugin.xml`. Here's an example

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.persistence_properties">
        <persistence-property name="datanucleus.magicBehaviour" value="false"
validator="org.datanucleus.properties.BooleanPropertyValidator"/>">
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above. Note also

- If you specify the attribute *datastore* as *true* in the `plugin.xml` then your property will be stored with the StoreManager (instead of NucleusContext where it would normally be stored).
- If you specify the attribute *manager-overrideable* as *true* in the `plugin.xml` then the user can specify this property additionally on the PM/EM (rather than just on the PMF/EMF)
- The *validator* attribute is optional but allows you to define a class that validates the values it can be set to.

6.1.2. Plugin Usage

Now you can access this property from within your DataNucleus plugin as follows

```
// from the NucleusContext
boolean magic = nucleusCtx.getConfiguration().getBooleanProperty
("datanucleus.magicBehaviour");

// or from the StoreManager
boolean magic = storeMgr.getBooleanProperty("datanucleus.magicBehaviour");
```

6.2. Store Manager

DataNucleus provides support for persisting objects to particular datastores. It provides this capability via a "Store Manager". It provides a Store Manager plugin for many datastores. You can extend DataNucleus's capabilities to support other datastores using the plugin extension **org.datanucleus.store_manager**.

6.2.1. Interface

If you want to implement support for another datastore you can achieve it by implementating the StoreManager interface. Javadoc.

For a brief guide on starting support for a new datastore, follow this HOWTO

6.2.2. Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file `plugin.xml` in your JAR at the root of the CLASSPATH. The file `plugin.xml` will look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.mystore" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_manager">
        <store-manager class-name="mydomain.MyStoreManager" url-key="mykey"
key="mykey"/>
    </extension>
</plugin>
```

Note that you also require a MANIFEST.MF file as described above.

6.2.3. Plugin Usage

The only thing remaining is to use your StoreManager. To do this you simply define your ConnectionURL to start with the *mykey* defined in the plugin spec (for example, with the RDBMS plugin, the connection URL starts *jdbc:db-name:...*. This will select your store manager based on that.

6.2.4. HOWTO Support A New Datastore

The process of adding support for a new datastore is simple in concept. It has particular components that can be supported or not. You don't have to support everything at once. The following gives guidelines on how to do it

- Is there a supported datastore that is similar in terms of its access ? If so, copy the existing datastore plugin (renaming packages etc). For example to support another OODB, you could take the "datanucleus-neodatis" plugin as a start point. Develop the XXXStoreManager class. **You are recommended to extend** *org.datanucleus.store.AbstractStoreManager*. Make sure you specify the *getSupportedOptions()* method specifying what is supported.

- Develop the ConnectionFactoryImpl. This provides connectivity to the datastore. If the datastore has a simple connection that you open then commit then this file will be very simple. Again, copy an existing one if there is something suitable. **You are recommended to extend** *org.datanucleus.store.connection.AbstractConnectionFactory*
- Develop the PersistenceHandler. This provides the capability to do insert/update/delete/fetch/locate and so provides support for persistence to this datastore. **You are recommended to extend** *org.datanucleus.store.AbstractPersistenceHandler*
- Does your datastore support the concept of a "schema" ? i.e you need to create some "table" in the datastore to store objects in? If so then you should implement the SchemaHandler. **You are recommended to extend** *org.datanucleus.store.schema.AbstractStoreSchemaHandler*
- Develop the JDOQL/JPQL support. Make use of the *org.datanucleus.query* classes. You need to decide how much of the query can be embodied in the native query language of the datastore. If you can't support some feature in the datastore native language then use the in-memory query evaluation
- Add on support for other features like inheritance strategy, discriminators, version checks, type converters as they are required.

As a guide, the basic connection, persistence could maybe be done in 2-4 days supporting "complete-table" only. To provide JDOQL or JPQL in-memory capability, you could do that in very short time also since the in-memory evaluator is done and the only thing you need to do is retrieve the candidate objects (of the candidate type). To provide JDOQL or JPQL in-datastore capability would take longer, how much would depend on the native query language structure for your datastore. Supporting things like other inheritance strategies may take significantly longer depending in the datastore.

6.3. ConnectionFactory

Any plugin for a datastore needs a way of connecting to the datastore and linking these connections into the persistence process. This is provided by way of a ConnectionFactory. This is extensible so you can define your own and register it for the datastore, and you use the plugin extension **org.datanucleus.store_connectionfactory**. This extension point is intended to be implemented by provider of the datastore plugin. A datastore will typically operate with 2 *ConnectionFactorys*, one for transactional operations and one for non-transactional operations.

6.3.1. Interface

Any Connection Factory plugin will need to implement *org.datanucleus.store.connection.ConnectionFactory*.  So you need to implement the following interface

```
public interface ConnectionFactory
{
    /**
     * Obtain a connection from the Factory.
     * The connection will be enlisted within the {@link org.datanucleus.Transaction}
associated to the ExecutionContext if "enlist" is set to true.
     * @param om the ObjectManager
     * @param options Any options for then creating the connection
     * @return the ManagedConnection
     */
    ManagedConnection getConnection(ExecutionContext ec, org.datanucleus.Transaction
transaction, Map options);

    /**
     * Create the ManagedConnection. Only used by ConnectionManager so do not call
this.
     * @param ec ExecutionContext (if any)
     * @param transactionOptions the Transaction options this connection will be
enlisted to, null if non existent
     * @return The ManagedConnection.
     */
    ManagedConnection createManagedConnection(ExecutionContext ec, Map
transactionOptions);
}
```

6.3.2. Plugin Specification

So we now have our custom "Connection Factory" and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. The file `plugin.xml` should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain.connectionfactory" name="My DataNucleus plug-in" provider-
name="MyCompany">
    <extension point="org.datanucleus.store_connectionfactory">
        <connectionfactory name="rdbms/tx" class-
name="org.datanucleus.store.rdbms.ConnectionFactoryImpl" transactional="true"
datastore="rdbms"/>
        <connectionfactory name="rdbms/nontx" class-
name="org.datanucleus.store.rdbms.ConnectionFactoryImpl" transactional="false"
datastore="rdbms"/>
    </extension>
</plugin>
```

Note that you also require a MANIFEST.MF file as described above.

So now for the datastore "rdbms" we will use this implementation when transactional or non-transactional

6.3.3. Lifecycle

The *ConnectionFactory* instance(s) are created when the StoreManager is instantiated and held as hard references during the lifecycle of the StoreManager.

6.4. AutoStart Mechanism

DataNucleus can discover the classes that it is managing at runtime, or you can define them in `persistence.xml`, or you can use an "auto-start" mechanism to inform DataNucleus of what classes it will be managing. DataNucleus provides 3 "auto-start" mechanisms, but also allows you to plugin your own variant using the plugin extension **org.datanucleus.autostart**.

Plugin extension-point	Key	Description	Location
org.datanucleus.autostart	classes	AutoStart mechanism specifying the list of classes to be managed	datanucleus-core
org.datanucleus.autostart	xml	AutoStart mechanism using an XML file to store the managed classes	datanucleus-core
org.datanucleus.autostart	schematable	AutoStart mechanism using a table in the RDBMS datastore to store the managed classes	datanucleus-rdbms

6.4.1. Interface

Any auto-start mechanism plugin will need to implement *org.datanucleus.store.AutoStartMechanism*. Javadoc So you need to implement the following interface

```
package org.datanucleus.store.autostart;

public interface AutoStartMechanism
{
    /** mechanism is disabled if None **/
    public static final String NONE = "None";

    /** mechanism is in Quiet mode **/
    public static final String MODE_QUIET = "Quiet";

    /** mechanism is in Checked mode **/
```

```
    public static final String MODE_CHECKED = "Checked";

    /** mechanism is in Ignored mode **/
    public static final String MODE_IGNORED = "Ignored";

    /**
     * Accessor for the mode of operation.
     * @return The mode of operation
     **/
    String getMode();

    /**
     * Mutator for the mode of operation.
     * @param mode The mode of operation
     **/
    void setMode(String mode);

    /**
     * Accessor for the data for the classes that are currently auto started.
     * @return Collection of {@link StoreData} elements
     * @throws DatastoreInitialisationException
     **/
    Collection getAllClassData() throws DatastoreInitialisationException;

    /**
     * Starts a transaction for writing (add/delete) classes to the auto start
mechanism.
     */
    void open();

    /**
     * Closes a transaction for writing (add/delete) classes to the auto start
mechanism.
     */
    void close();

    /**
     * Whether it's open for writing (add/delete) classes to the auto start mechanism.
     * @return whether this is open for writing
     */
    public boolean isOpen();

    /**
     * Method to add a class/field (with its data) to the currently-supported list.
     * @param data The data for the class.
     **/
    void addClass(StoreData data);

    /**
     * Method to delete a class/field that is currently listed as supported in
     * the internal storage.
```

```
     * It does not drop the schema of the DatastoreClass
     * neither the contents of it. It only removes the class from the
     * AutoStart mechanism.
     * TODO Rename this method to allow for deleting fields
     * @param name The name of the class/field
     **/
    void deleteClass(String name);

    /**
     * Method to delete all classes that are currently listed as supported in
     * the internal storage. It does not drop the schema of the DatastoreClass
     * neither the contents of it. It only removes the classes from the
     * AutoStart mechanism.
     **/
    void deleteAllClasses();

    /**
     * Utility to return a description of the storage for this mechanism.
     * @return The storage description.
     **/
    String getStorageDescription();
}
```

You can extend *org.datanucleus.store.AbstractAutoStartMechanism*.

6.4.2. Implementation

So lets assume that you want to create your own auto-starter *MyAutoStarter*.

```
package mydomain;

import org.datanucleus.store.AutoStartMechanism;
import org.datanucleus.store.AbstractAutoStartMechanism;
import org.datanucleus.store.StoreManager;
import org.datanucleus.ClassLoaderResolver;

public class MyAutoStarter extends AbstractAutoStartMechanism
{
    public MyAutoStarter(StoreManager storeMgr, ClassLoaderResolver clr)
    {
        super();
    }

    ... (implement the required methods)
}
```

6.4.3. Plugin Specification

When we have defined our "AutoStartMechanism" we just need to make it into a DataNucleus

plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. The file `plugin.xml` should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.autostart">
        <autostart name="myStarter" class-name="mydomain.MyAutoStarter"/>
    </extension>
</plugin>
```

Note that you also require a MANIFEST.MF file as described above.

6.4.4. Plugin Usage

The only thing remaining is to use your new *AutoStartMechanism* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the persistence property **datanucleus.autoStartMechanism** to *myStarter* (the name you specified in the `plugin.xml` file).

6.5. Identifier NamingFactory

Mapping a persistable class to the datastore requires specification of naming of datastore table/column (as well as constraint) identifiers. If the user doesn't define names for such identifiers then defaults have to be provided. This is where we have a *NamingFactory*. DataNucleus provides its own internal naming factories (for JDO, and for JPA), but also allows you to plugin your own naming factory.

Note that currently RDBMS doesn't use this extension and instead uses the RDBMS Identifier Factory extension.

You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.identifier_namingfactory**.

Plugin extension-point	Key	Description	Location
org.datanucleus.identifier_namingfactory	datanucleus2	NamingFactory providing DataNucleus 2+ namings	datanucleus-core
org.datanucleus.identifier_namingfactory	jpa	NamingFactory providing JPA-compliant namings	datanucleus-core

6.5.1. Interface

Any identifier factory plugin will need to implement *org.datanucleus.store.schema.naming.NamingFactory* Javadoc. So you need to implement the following interface

```
package org.datanucleus.store;

public interface IdentifierFactory
{
    /**
     * Method to set the provided list of keywords as names that identifiers have to
surround by quotes to use.
     * @param keywords The keywords
     * @return This naming factory
     */
    NamingFactory setReservedKeywords(Set<String> keywords);

    /**
     * Method to set the maximum length of the name of the specified schema component.
     * @param cmpt The component
     * @param max The maximum it accepts
     * @return This naming factory
     */
    NamingFactory setMaximumLength(SchemaComponent cmpt, int max);

    /**
     * Method to set the quote string to use (when the identifiers need to be quoted).
     * See <pre>setIdentifierCase</pre>.
     * @param quote The quote string
     * @return This naming factory
     */
    NamingFactory setQuoteString(String quote);

    /**
     * Method to set the word separator of the names.
     * @param sep Separator
     * @return This naming factory
     */
    NamingFactory setWordSeparator(String sep);

    /**
     * Method to set the required case of the names.
     * @param nameCase Required case
     * @return This naming factory
     */
    NamingFactory setNamingCase(NamingCase nameCase);

    /**
     * Method to return the name of the table for the specified class.
     * @param cmd Metadata for the class
     * @return Name of the table
     */
    String getTableName(AbstractClassMetaData cmd);

    /**
```

```
     * Method to return the name of the (join) table for the specified field.
     * @param mmd Metadata for the field/property needing a join table
     * @return Name of the table
     */
    String getTableName(AbstractMemberMetaData mmd);

    /**
     * Method to return the name of the column for the specified class (version,
datastore-id, discriminator etc).
     * @param cmd Metadata for the class
     * @param type Column type
     * @return Name of the column
     */
    String getColumnName(AbstractClassMetaData cmd, ColumnType type);

    /**
     * Method to return the name of the column for the specified field.
     * If you have multiple columns for a field then call the other
<pre>getColumnName</pre> method.
     * @param mmd Metadata for the field
     * @param type Type of column
     * @return The column name
     */
    String getColumnName(AbstractMemberMetaData mmd, ColumnType type);

    /**
     * Method to return the name of the column for the position of the specified
field.
     * Normally the position will be 0 since most fields map to a single column, but
where you have a FK
     * to an object with composite id, or where the Java type maps to multiple columns
then the position is used.
     * @param mmd Metadata for the field
     * @param type Type of column
     * @param position Position of the column
     * @return The column name
     */
    String getColumnName(AbstractMemberMetaData mmd, ColumnType type, int position);

    /**
     * Method to return the name of the column for the position of the specified
EMBEDDED field, within the specified owner field.
     * For example, say we have a class Type1 with field "field1" that is marked as
embedded, and this is of type Type2.
     * In turn Type2 has a field "field2" that is also embedded, of type Type3. Type3
has a field "name". So to get the column name for
     * Type3.name in the table for Type1 we call "getColumnName({mmdForField1InType1,
mmdForField2InType2, mmdForNameInType3}, 0)".
     * @param mmds MetaData for the field(s) with the column. The first value is the
original field that is embedded, followed by fields of the embedded object(s).
     * @param position The position of the column (where this field has multiple
```

```
columns)
     * @return The column name
     */
    String getColumnName(List<AbstractMemberMetaData> mmds, int position);

    /**
     * Method to return the name of an index specified at class level.
     * @param cmd Metadata for the class
     * @param idxmd The index metadata
     * @param position Number of the index at class level (first is 0)
     * @return Name of the index
     */
    String getIndexName(AbstractClassMetaData cmd, IndexMetaData idxmd, int position);

    /**
     * Method to return the name of an index specified at member level.
     * @param mmd Metadata for the member
     * @param idxmd The index metadata
     * @return Name of the index
     */
    String getIndexName(AbstractMemberMetaData mmd, IndexMetaData idxmd);

    // TODO Support foreign-key naming

    /**
     * Method to return the name of sequence.
     * @param seqmd Metadata for the sequence
     * @return Name of the sequence
     */
    String getSequenceName(SequenceMetaData seqmd);
}
```

Be aware that you can extend *org.datanucleus.store.schema.naming.AbstractNamingFactory* Javadoc.

6.5.2. Implementation

Let's assume that you want to provide your own identifier factory *MyNamingFactory*

```
package mydomain;

import org.datanucleus.store.schema.naming.AbstractNamingFactory

public class MyIdentifierFactory extends AbstractNamingFactory
{
    /**
     * Constructor.
     * @param nucCtx NucleusContext
     */
    public MyNamingFactory(NucleusContext nucCtx)
    {
        super(nucCtx);
        ...
    }

    .. (implement the rest of the interface)
}
```

6.5.3. Plugin Specification

When we have defined our "NamingFactory" we just need to make it into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root. This file should look like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.identifier_namingfactory">
        <identifierfactory name="myfactory" class-name="mydomain.MyNamingFactory"/>
    </extension>
</plugin>
```

Note that you also require a MANIFEST.MF file as described above.

6.5.4. Plugin Usage

The only thing remaining is to use your new *NamingFactory* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the PMF property *datanucleus.identifier.namingFactory* to *myfactory* (the name you specified in the `plugin.xml` file).

6.6. ClassLoader Resolver

DataNucleus allows specification of a class-loader resolver at runtime. It provides its own "built-in" resolver (*org.datanucleus.ClassLoaderResolverImpl*) that matches the requirements of the JDO spec (and following what the JPA spec needs), but this capability is extensible. You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.classloader_resolver**.

6.6.1. Built-in Implementation

At runtime, DataNucleus will need to load the classes being persisted. To do this it needs to utilise a particular class loading mechanism. The JDO specification defines how class-loading is to operate in a JDO implementation and this is what DataNucleus uses by default. In brief the JDO class-loading mechanism utilises 3 class loaders

- When creating the PersistenceManagerFactory you can specify a class loader. This is used first if specified
- The second class loader to try is the class loader for the current thread.
- The third class loader to try is the class loader for the PMF context.

If a class cannot be loaded using these three loaders then an exception is typically thrown depending on the operation.

6.6.2. Interface

Any identifier factory plugin will need to implement *org.datanucleus.ClassLoaderResolver*. Javadoc. So you need to implement the following interface

```
package org.datanucleus;

public interface ClassLoaderResolver
{
    /**
     * Class loading method, allowing specification of a primary loader.
     * This method does not initialize the class
     * @param name Name of the Class to be loaded
     * @param primary the primary ClassLoader to use (or null)
     * @return The Class given the name, using the specified ClassLoader
     * @throws ClassNotResolvedException if the class can't be found in the classpath
     */
    public Class classForName(String name, ClassLoader primary);

        /**
     * Class loading method, allowing specification of a primary loader
     * and whether the class should be initialised or not.
     * @param name Name of the Class to be loaded
     * @param primary the primary ClassLoader to use (or null)
     * @param initialize whether to initialize the class or not.
     * @return The Class given the name, using the specified ClassLoader
     * @throws ClassNotResolvedException if the class can't be found in the classpath
     */
    public Class classForName(String name, ClassLoader primary, boolean initialize);

    /**
     * Class loading method. This method does not initialize the class
     * @param name Name of the Class to be loaded
     * @return The Class given the name, using the specified ClassLoader
```

```
     */
    public Class classForName(String name);

    /**
     * Class loading method, allowing for initialisation of the class.
     * @param name Name of the Class to be loaded
     * @param initialize whether to initialize the class or not.
     * @return The Class given the name, using the specified ClassLoader
     */
    public Class classForName(String name, boolean initialize);

    /**
     * Method to test whether the type represented by the specified class_2
     * parameter can be converted to the type represented by class_name_1 parameter.
     * @param class_name_1 Class name
     * @param class_2 Class to compare against
     * @return Whether they are assignable
     */
    public boolean isAssignableFrom(String class_name_1, Class class_2);

    /**
     * Method to test whether the type represented by the specified class_name_2
     * parameter can be converted to the type represented by class_1 parameter.
     * @param class_1 First class
     * @param class_name_2 Class name to compare against
     * @return Whether they are assignable
     */
    public boolean isAssignableFrom(Class class_1, String class_name_2);

    /**
     * Method to test whether the type represented by the specified class_name_2
     * parameter can be converted to the type represented by class_name_1 parameter.
     * @param class_name_1 Class name
     * @param class_name_2 Class name to compare against
     * @return Whether they are assignable
     */
    public boolean isAssignableFrom(String class_name_1, String class_name_2);

    /**
     * ClassLoader registered to load classes created at runtime. One ClassLoader can
     * be registered, and if one ClassLoader is already registered, the registered
ClassLoader
     * is replaced by <code>loader</code>.
     * @param loader The ClassLoader in which classes are defined
     */
    public void registerClassLoader(ClassLoader loader);

    /**
     * ClassLoader registered by users to load classes. One ClassLoader can
     * be registered, and if one ClassLoader is already registered, the registered
ClassLoader
```

```
     * is replaced by <code>loader</code>.
     * @param loader The ClassLoader in which classes are loaded
     */
    public void registerUserClassLoader(ClassLoader loader);

    /**
     * Finds all the resources with the given name.
     * @param resourceName the resource name. If <code>resourceName</code> starts with
"/",
     *           remove it before searching.
     * @param primary the primary ClassLoader to use (or null)
     * @return An enumeration of URL objects for the resource. If no resources could
be found,
     *           the enumeration will be empty.
     * Resources that the class loader doesn't have access to will not be in the
enumeration.
     * @throws IOException If I/O errors occur
     * @see ClassLoader#getResources(java.lang.String)
     */
    public Enumeration getResources(String resourceName, ClassLoader primary) throws
IOException;

    /**
     * Finds the resource with the given name.
     * @param resourceName the path to resource name relative to the classloader root
path.
     *        If <code>resourceName</code> starts with "/", remove it.
     * @param primary the primary ClassLoader to use (or null)
     * @return A URL object for reading the resource, or null if the resource could
not be found or
     *         the invoker doesn't have adequate privileges to get the resource.
     * @throws IOException If I/O errors occur
     * @see ClassLoader#getResource(java.lang.String)
     */
    public URL getResource(String resourceName, ClassLoader primary);

    /**
     * Sets the primary classloader for the current thread.
     * The primary should be kept in a ThreadLocal variable.
     * @param primary the primary classloader
     */
    void setPrimary(ClassLoader primary);

    /**
     * Unsets the primary classloader for the current thread
     */
    void unsetPrimary();
}
```

Be aware that you can extend *org.datanucleus.ClassLoaderResolverImpl* if you just want to change

some behaviour of the default loader process. Your class loader resolver should provide a constructor taking an argument of type *ClassLoader* which will be the loader that the PM/EM is using at initialisation (your class can opt to not use this, but must provide the constructor)

6.6.3. Implementation

Let's suppose you want to provide your own resolver *MyClassLoaderResolver*

```
package mydomain;

import org.datanucleus.ClassLoaderResolver;

public class MyClassLoaderResolver implements ClassLoaderResolver
{
    /**
     * Constructor for PersistenceManager cases.
     * @param pmLoader Loader from PM initialisation time.
     */
    public MyClassLoaderResolver(ClassLoader pmLoader)
    {
        ...
    }

    .. (implement the interface)
}
```

6.6.4. Plugin Specification

When we have defined our "IdentifierFactory" we just need to make it into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.classloader_resolver">
        <class-loader-resolver name="myloader" class-
name="mydomain.MyClassLoaderResolver"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

6.6.5. Plugin Usage

The only thing remaining is to use your new *ClassLoaderResolver* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the PMF property *datanucleus.classLoaderResolverName* to *myloader* (the name you specified in the `plugin.xml` file).

6.7. Value Generators

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the generation of identity or field values. DataNucleus provides a large selection of generators but is structured so that you can easily add your own variant and have it usable within your DataNucleus usage. Below are listed some of those available, but each store plugin typically will define its own. The JDO/JPA specs define various that are required. You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.store_valuegenerator**.

Plugin extension-point	Key	Datastore	Description	Location
org.datanucleus.store_valuegenerator	auid	all datastores	Value Generator using AUIDs	datanucleus-core
org.datanucleus.store_valuegenerator	uuid-hex	all datastores	Value Generator using uuid-hex	datanucleus-core
org.datanucleus.store_valuegenerator	uuid-string	all datastores	Value Generator using uuid-string	datanucleus-core
org.datanucleus.store_valuegenerator	timestamp	all datastores	Value Generator using Timestamp	datanucleus-core
org.datanucleus.store_valuegenerator	timestamp-value	all datastores	Value Generator using Timestamp millisecs value	datanucleus-core
org.datanucleus.store_valuegenerator	increment	rdbms	Value Generator using increment strategy	datanucleus-rdbms
org.datanucleus.store_valuegenerator	sequence	rdbms	Value Generator using datastore sequences	datanucleus-rdbms
org.datanucleus.store_valuegenerator	table-sequence	rdbms	Value Generator using a database table to generate sequences (same as increment)	datanucleus-rdbms
org.datanucleus.store_valuegenerator	max	rdbms	Value Generator using max(COL)+1 strategy	datanucleus-rdbms
org.datanucleus.store_valuegenerator	datastore-uuid-hex	rdbms	Value Generator using uuid-hex attributed by the datastore	datanucleus-rdbms
org.datanucleus.store_valuegenerator	identity	mongodb	Value Generator for MongoDB using identity strategy	datanucleus-mongodb
org.datanucleus.store_valuegenerator	identity	neo4j	Value Generator for Neo4j using identity strategy	datanucleus-neo4j

The following sections describe how to create your own value generator plugin for DataNucleus.

6.7.1. Interface

Any value generator plugin will need to implement *org.datanucleus.store.valuegenerator.ValueGenerator* Javadoc. So you need to implement the following interface

```
public interface ValueGenerator
{
    String getName ();

    void allocate (int additional);

    Object next ();
    Object current ();

    long nextValue();
    long currentValue();
}
```

6.7.2. Implementation

DataNucleus provides an abstract base class *org.datanucleus.store.valuegenerator.AbstractValueGenerator* to extend if you don't require datastore access. If you do require (RDBMS) datastore access for your ValueGenerator then you can extend *org.datanucleus.store.rdbms.valuegenerator.AbstractRDBMSValueGenerator* Let's give an example, here we want a generator that provides a form of UUID identity. We define our class as

```
package mydomain;

import org.datanucleus.store.valuegenerator.ValueGenerationBlock;
import org.datanucleus.store.valuegenerator.AbstractValueGenerator;

public class MyUUIDValueGenerator extends AbstractValueGenerator
{
    public MyUUIDValueGenerator(String name, Properties props)
    {
        super(name, props);
    }

    /**
     * Method to reserve "size" ValueGenerations to the ValueGenerationBlock.
     * @param size The block size
     * @return The reserved block
     */
    public ValueGenerationBlock reserveBlock(long size)
    {
        Object[] ids = new Object[(int) size];
        for (int i = 0; i < size; i++)
        {
            ids[i] = getIdentifier();
        }
        return new ValueGenerationBlock(ids);
    }

    /**
     * Create a UUID identifier.
     * @return The identifier
     */
    private String getIdentifier()
    {
        ... Write this method to generate the identifier
    }
}
```

As show you need a constructor taking 2 arguments *String* and *java.util.Properties*. The first being the name of the generator, and the second containing properties for use in the generator.

- *class-name* Name of the class that the value is being added to
- *root-class-name* Name of the root class in this inheritance tree
- *field-name* Name of the field whose value is being set (not provided if this is datastore identity field)
- *catalog-name* Catalog that objects of the class are stored in
- *schema-name* Schema that objects of the class are stored in
- *table-name* Name of the (root) table storing this field

- *column-name* Name of the column storing this field
- *sequence-name* Name of the sequence (if specified in the MetaData)

6.7.3. Plugin Specification

So we now have our custom "value generator" and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store_valuegenerator">
        <valuegenerator name="myuuid" class-name="mydomain.MyUUIDValueGenerator"
unique="true"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

The name "myuuid" is what you will use as the "strategy" when specifying to use it in MetaData. The flag "unique" is only needed if your generator is to be unique across all requests. For example if your generator was only unique for a particular class then you should omit that part. Thats all. You now have a DataNucleus "value generator" plugin.

6.7.4. Plugin Usage

To use your value generator you would reference it in your JDO MetaData like this

```
<class name="MyClass">
    <datastore-identity strategy="myuuid"/>
    ...
</class>
```

Don't forget that if you write a value generator that could be of value to others you could easily donate it to DataNucleus for inclusion in the next release.

6.8. InMemory Query Methods

JDOQL/JPQL are defined to support particular methods/functions as part of the supported syntax. This support is provided by way of an extension point, with support for these methods/functions added via extensions. You can make use of this extension point to add on your own methods/functions - obviously this will be DataNucleus specific.

This plugin extension point is currently only for evaluation of queries in-memory. It will have no effect where the query is evaluated in the datastore. The plugin extension used here is **org.datanucleus.query_method_evaluators**.

6.8.1. Interface

Any query method/function plugin will need to implement *org.datanucleus.query.evaluator.memory.InvocationEvaluator*. Javadoc. So you need to implement the following interface

```
public interface InvocationEvaluator
{
    /**
     * Method to evaluate the InvokeExpression, as part of the overall evaluation
     * defined by the InMemoryExpressionEvaluator.
     * @param expr The expression for invocation
     * @param invokedValue Value on which we are invoking this method
     * @param eval The overall evaluator for in-memory
     * @return The result
     */
    Object evaluate(InvokeExpression expr, Object invokedValue,
InMemoryExpressionEvaluator eval);
}
```

6.8.2. Implementation

Let's assume that you want to provide your own method for "String" _toUpperCase : obviously this is provided out of the box, but is here as an example.

```
public class StringToUpperCaseMethodEvaluator implements InvocationEvaluator
{
    public Object evaluate(InvokeExpression expr, Object invokedValue,
InMemoryExpressionEvaluator eval)
    {
        String method = expr.getOperation(); // Will be "toUpperCase"

        if (invokedValue == null)
        {
            return null;
        }
        if (!(invokedValue instanceof String))
        {
            throw new NucleusException(Localiser.msg("021011", method, invokedValue
.getClass().getName()));
        }
        return ((String)invokedValue).toUpperCase();
    }
}
```

6.8.3. Plugin Specification

When we have defined our query method we just need to make it into a DataNucleus plugin. To do

this you simply add a file `plugin.xml` to your JAR at the root, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.query_method_evaluators">
        <query-method-evaluator class="java.lang.String" method="toUpperCase"
evaluator="org.datanucleus.query.evaluator.memory.StringToUpperCaseMethodEvaluator"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

6.8.4. Plugin Usage

The only thing remaining is to use your method in a JDOQL/JPQL query, like this

```
Query q = pm.newQuery("SELECT FROM mydomain.Product WHERE name.toUpperCase() ==
'KETTLE'");
```

so when evaluating the query in memory it will call this evaluator class for the field 'name'.

6.9. TransactionManagerLocator

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the locator for JTA TransactionManagers (since J2EE doesnt define a standard mechanism for location). DataNucleus provides several plugins for the principal application servers available but is structured so that you can easily add your own variant and have it usable within your DataNucleus usage.

Locators for JTA TransactionManagers can be plugged using the plugin extension **org.datanucleus.jta_locator**. These are of relevance when running with JTA transaction and linking in to the JTA transaction of some controlling application server.

Plugin extension-point	Key	Description	Location
org.datanucleus.jta_locator	jboss	JBoss	datanucleus-core
org.datanucleus.jta_locator	jonas	JOnAS	datanucleus-core
org.datanucleus.jta_locator	jotm	JOTM	datanucleus-core
org.datanucleus.jta_locator	oc4j	OC4J	datanucleus-core
org.datanucleus.jta_locator	orion	Orion	datanucleus-core
org.datanucleus.jta_locator	resin	Resin	datanucleus-core
org.datanucleus.jta_locator	sap	SAP app server	datanucleus-core
org.datanucleus.jta_locator	sun	Sun ONE app server	datanucleus-core

Plugin extension-point	Key	Description	Location
org.datanucleus.jta_locator	weblogic	WebLogic app server	datanucleus-core
org.datanucleus.jta_locator	websphere	WebSphere 4/5	datanucleus-core
org.datanucleus.jta_locator	custom_jndi	Custom JNDI	datanucleus-core
org.datanucleus.jta_locator	atomikos	Atomikos	datanucleus-core

The following sections describe how to create your own JTA Locator plugin for DataNucleus.

6.9.1. Interface

If you have your own JTA Locator you can easily use it with DataNucleus. DataNucleus defines a TransactionManagerLocator interface and you need to implement this. Javadoc.

```
package org.datanucleus.jta;

import javax.transaction.TransactionManager;
import org.datanucleus.ClassLoaderResolver;

public interface TransactionManagerLocator
{
    /**
     * Method to return the TransactionManager.
     * @param clr ClassLoader resolver
     * @return The TransactionManager
     */
    TransactionManager getTransactionManager(ClassLoaderResolver clr);
}
```

So you need to create a class, *MyTransactionManagerLocator* for example, that implements this interface.

6.9.2. Plugin Specification

Once you have this implementation you then need to make the class available as a DataNucleus plugin. You do this by putting a file `plugin.xml` in your JAR at the root of the CLASSPATH, like this

```
<?xml version="1.0"?>
<plugin id="mydomain.mylocator" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.jta_locator">
        <jta_locator name="MyLocator" class-
name="mydomain.MyTransactionManagerLocator"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

6.9.3. Plugin Usage

The only thing remaining is to use your JTA Locator plugin. To do this you specify the persistence property **datanucleus.transaction.jta.transactionManagerLocator** as *MyLocator* (the "name" in `plugin.xml`).

Chapter 7. RDBMS ExtensionPoints

7.1. RDBMS Java Mapping

When persisting a class to an RDBMS datastore there is a *mapping* process from class/field to table/column. DataNucleus provides a mapping process and allows users to define their own mappings where required. Each field is of a particular type, and where a field is to be persisted as a second-class object you need to define a mapping. DataNucleus defines mappings for all of the required JDO/JPA types but you want to persist some of your own types as second-class objects. This extension is not required for other datastores, just RDBMS.

A *type mapping* defines the way to convert between an object of the Java type and its datastore representation (namely a column or columns in a datastore table). There are 2 types of Java types that can be mapped. These are *mutable* (something that can be updated, such as java.util.Date) and *immutable* (something that is fixed from the point of construction, such as java.awt.Color).

7.1.1. Java type to single column mapping

Creating a Mapping class is optional and as long as you have defined a TypeConverter then this will be mapped automatically using *TypeConverterMapping*. For 99.9% of types it should not be necessary to define a JavaTypeMapping as well. If you really want to add a mapping then just create an implementation of JavaTypeMapping, extending SingleFieldMapping (find an example in GitHub and copy it).

7.1.2. Java Type to multiple column mapping

As we mentioned for the Type Converter extension you can provide a multi-column TypeConverter for this situation. You only need a JavaTypeMapping if you want to provide for RDBMS method invocation. The guide below shows you how to do write this mapping.

The simplest way to describe how to define your own mapping is to give an example. Here we'll use the example of the Java AWT class *Color*. This has 3 colour components (red, green, and blue) as well as an alpha component. So here we want to map the Java type java.awt.Color to 4 datastore columns - one for each of the red, green, blue, and alpha components of the colour. To do this we define a mapping class extending the DataNucleus class *org.datanucleus.store.rdbms.mapping.java.SingleFieldMultiMapping* Javadoc.

```
package org.mydomain;

import org.datanucleus.PersistenceManager;
import org.datanucleus.metadata.AbstractPropertyMetaData;
import org.datanucleus.store.rdbms.mapping.java.SingleFieldMultiMapping;
import org.datanucleus.store.rdbms.mapping.java.JavaTypeMapping;
import org.datanucleus.store.rdbms.table.Table;

public class ColorMapping extends SingleFieldMultiMapping
{
```

```
    /**
     * Initialize this JavaTypeMapping with the given DatastoreAdapter for the given
FieldMetaData.
     * @param dba The Datastore Adapter that this Mapping should use.
     * @param fmd FieldMetaData for the field to be mapped (if any)
     * @param table The table holding this mapping
     * @param clr the ClassLoaderResolver
     */
    public void initialize(AbstractMemberMetaData mmd, Table table,
ClassLoaderResolver clr)
    {
        super.initialize(mmd, table, clr);

        addDatastoreField(ClassNameConstants.INT); // Red
        addDatastoreField(ClassNameConstants.INT); // Green
        addDatastoreField(ClassNameConstants.INT); // Blue
        addDatastoreField(ClassNameConstants.INT); // Alpha
    }

    public Class getJavaType()
    {
        return Color.class;
    }

    public Object getSampleValue(ClassLoaderResolver clr)
    {
        return java.awt.Color.red;
    }

    public void setObject(ExecutionContext ec, PreparedStatement ps, int[] exprIndex,
Object value)
    {
        Color color = (Color) value;
        if (color == null)
        {
            getColumnMapping(0).setObject(ps, exprIndex[0], null);
            getColumnMapping(1).setObject(ps, exprIndex[1], null);
            getColumnMapping(2).setObject(ps, exprIndex[2], null);
            getColumnMapping(3).setObject(ps, exprIndex[3], null);
        }
        else
        {
            getColumnMapping(0).setInt(ps, exprIndex[0], color.getRed());
            getColumnMapping(1).setInt(ps, exprIndex[1], color.getGreen());
            getColumnMapping(2).setInt(ps, exprIndex[2], color.getBlue());
            getColumnMapping(3).setInt(ps, exprIndex[3], color.getAlpha());
        }
    }

    public Object getObject(ExecutionContext ec, ResultSet rs, int[] exprIndex)
    {
```

```
        try
        {
            // Check for null entries
            if (((ResultSet)rs).getObject(exprIndex[0]) == null)
            {
                return null;
            }
        }
        catch (Exception e)
        {
            // Do nothing
        }

        int red = getColumnMapping(0).getInt(rs, exprIndex[0]);
        int green = getColumnMapping(1).getInt(rs, exprIndex[1]);
        int blue = getColumnMapping(2).getInt(rs, exprIndex[2]);
        int alpha = getColumnMapping(3).getInt(rs, exprIndex[3]);
        return new Color(red,green,blue,alpha);
    }
}
```

In the initialize() method we've created 4 columns - one for each of the red, green, blue, alpha components of the colour. The argument passed in when constructing these columns is the Java type name of the column data being stored. The other 2 methods of relevance are the setObject() and getObject(). These have the task of mapping between the *Color* object and its datastore representation (the 4 columns). That's all there is to it.

The only thing we need to do is enable use of this Java type when running DataNucleus. To do this we create a `plugin.xml` (at the root of our CLASSPATH) to contain our mappings.

```
<?xml version="1.0"?>
<plugin>
    <extension point="org.datanucleus.store.rdbms.java_mapping">
        <mapping java-type="java.awt.Color" mapping-
class="org.mydomain.MyColorMapping"/>
    </extension>
</plugin>
```

Note that we also require a `MANIFEST.MF` file as described above. When using the DataNucleus Enhancer, SchemaTool or runtime, DataNucleus automatically searches for the *mapping definition* at `/plugin.xml` files in the CLASSPATH.

Obviously, since DataNucleus already supports *java.awt.Color* there is no need to add this particular mapping to DataNucleus yourself, but this demonstrates the way you should do it for any type you wish to add.

If your Java type that you want to map maps direct to a single column then you would instead extend org.datanucleus.store.mapping.java.SingleFieldMapping and wouldn't need to add the columns yourself. Look at datanucleus-rdbms for many examples of doing it this way.

7.2. RDBMS Column Mapping

When persisting a class to an RDBMS datastore there is a *mapping* process from class/field to table/column. The most common thing to configure is the JavaTypeMapping so that it maps between the java type and the column(s) in the desired way. We can also configure the mapping to the datastore type. So, for example, we define what JDBC types we can map a particular Java type to. By "datastore type" we mean the JDBC type, such as BLOB, INT, VARCHAR etc. DataNucleus provides column mappings for the vast majority of JDBC types and the handlings will almost always be adequate. What you could do though is make a particular Java type persistable using a different JDBC type if you wished.

7.2.1. Interface

To define your own column mapping you need to implement *org.datanucleus.store.rdbms.mapping.column.ColumnMapping* Javadoc.

So you can define how to convert the datastore column value to/from common Java types. Look at datanucleus-rdbms for examples. Note that your XXXColumnMapping should have a single constructor taking in *JavaTypeMapping mapping, StoreManager storeMgr, Column col.*

7.2.2. Plugin Specification

To give an example of what the plugin specification looks like

```
<?xml version="1.0"?>
<plugin id="mydomain.myplugins" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.store.rdbms.column_mapping">
        <mapping java-type="java.lang.Character" column-mapping-
class="org.datanucleus.store.rdbms.mapping.column.CharColumnMapping"
            jdbc-type="CHAR" sql-type="CHAR" default="true"/>
        <mapping java-type="java.lang.Character" column-mapping-
class="org.datanucleus.store.rdbms.mapping.column.IntegerColumnMapping"
            jdbc-type="INTEGER" sql-type="INT" default="false"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above. So in this definition we have defined that a field of type "Character" can be mapped to JDBC type of CHAR or INTEGER (with CHAR the default)

7.2.3. Missing JDBC definitions

While DataNucleus attempts to provide all useful JDBC type mappings out of the box, occasionally you may come across one that is not defined, even though we provide a ColumnMapping class for that JDBC type. For example, if you got this error message

```
    JDBC type XXX declared for field "org.datanucleus.test.MyClass.yyyField" of java
type java.lang.YYY cant be mapped for this datastore.
```

This means that while the JDBC driver may support this JDBC type, we haven't defined it for the java type "YYY" in the *plugin.xml* file. All that you need to do is add a `plugin.xml`

```
<?xml version="1.0"?>
<plugin id="mydomain.myplugins" name="DataNucleus plug-ins" provider-name="My
Company">
    <extension point="org.datanucleus.store.rdbms.column_mapping">
        <mapping java-type="java.lang.YYY" column-mapping-
class="org.datanucleus.store.rdbms.mapping.column.XXXColumnMapping"
            jdbc-type="XXX" sql-type="XXX" default="false"/>
    </extension>
</plugin>
```

together with `MANIFEST.MF` to your application, and it will be supported. Obviously if it is of a type that you think we ought to support out-of-the-box then raise an issue against the datanucleus-rdbms with your `plugin.xml` definition and mention the datastore it was needed for.

7.3. RDBMS Datastore Adapter

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the adapter for the datastore. The datastore adapter provides the translation between DataNucleus and the specifics of the RDBMS in use. DataNucleus provides support for a large selection of RDBMS but is structured so that you can easily add your own adapter for your RDBMS and have it usable within your DataNucleus usage.

DataNucleus supports many RDBMS databases, and by default, ALL RDBMS are supported without the need to extend DataNucleus. Due to incompabilities, or specifics of each RDBMS database, it's allowed to extend DataNucleus to make the support to a specific database fit better to DataNucleus and your needs. The RDBMS page lists all RDBMS databases that have been tested with DataNucleus, and some of these databases has been adapted internally to get a good fit. You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.store.rdbms.datastoreadapter**.

Plugin extension-point	Key	Description	Location
org.datanucleus.store.rdbms.datastoreadapter	derby	Adapter for Apache Derby	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	db2	Adapter for IBM DB2	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	as/400	Adapter for IBM DB2 AS/400	datanucleus-rdbms

Plugin extension-point	Key	Description	Location
org.datanucleus.store.rdbms.datastoreadapter	firebird	Adapter for Firebird/Interbase	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	microsoft	Adapter for SQLServer	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	h2	Adapter for H2	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	hsql	Adapter for HSQLDB	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	mysql	Adapter for MySQL	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	sybase	Adapter for Sybase	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	oracle	Adapter for Oracle	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	pointbase	Adapter for Pointbase	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	postgresql	Adapter for PostgreSQL	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	sapdb	Adapter for SAPDB/MaxDB	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	sqlite	Adapter for SQLite	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	timesten	Adapter for Timesten	datanucleus-rdbms
org.datanucleus.store.rdbms.datastoreadapter	informix	Adapter for Informix	datanucleus-rdbms

DataNucleus supports a very wide range of RDBMS datastores. It will typically auto-detect the datastore adapter to use and select it. This, in general, will work well and the user will not need to change anything to benefit from this behaviour. There are occasions however where a user may need to provide their own datastore adapter and use that. For example if their RDBMS is a new version and something has changed relative to the previous (supported) version, or where the auto-detection fails to identify the adapter since their RDBMS is not yet on the supported list.

By default when you create a PMF to connect to a particular datastore DataNucleus will automatically detect the *datastore adapter* to use and will use its own internal adapter for that type of datastore. The default behaviour is overridden using the persistence property **datanucleus.rdbms.datastoreAdapterClassName**, which specifies the class name of the datastore adapter class to use. This class must implement the DataNucleus interface *DatastoreAdapter* Javadoc.

So you need to implement *DatastoreAdapter*. You have 2 ways to go here. You can either start from scratch (when writing a brand new adapter), or you can take the existing DataNucleus adapter for a

particular RDBMS and change (or extend) it. Let's take an example so you can see what is typically included in such an Adapter. Bear in mind that ALL RDBMS are different in some (maybe small) way, so you may have to specify very little in this adapter, or you may have a lot to specify depending on the RDBMS, and how capable it's JDBC drivers are.

```
public class MySQLAdapter extends BaseDatastoreAdapter
{
    /**
     * A string containing the list of MySQL keywords that are not also SQL/92
     * <i>reserved words</i>, separated by commas.
     */
    public static final String NONSQL92_RESERVED_WORDS =
        "ANALYZE,AUTO_INCREMENT,BDB,BERKELEYDB,BIGINT,BINARY,BLOB,BTREE," +
        "CHANGE,COLUMNS,DATABASE,DATABASES,DAY_HOUR,DAY_MINUTE,DAY_SECOND," +
        "DELAYED,DISTINCTROW,DIV,ENCLOSED,ERRORS,ESCAPED,EXPLAIN,FIELDS," +
        "FORCE,FULLTEXT,FUNCTION,GEOMETRY,HASH,HELP,HIGH_PRIORITY," +
        "HOUR_MINUTE,HOUR_SECOND,IF,IGNORE,INDEX,INFILE,INNODB,KEYS,KILL," +
        "LIMIT,LINES,LOAD,LOCALTIME,LOCALTIMESTAMP,LOCK,LONG,LONGBLOB," +
        "LONGTEXT,LOW_PRIORITY,MASTER_SERVER_ID,MEDIUMBLOB,MEDIUMINT," +
        "MEDIUMTEXT,MIDDLEINT,MINUTE_SECOND,MOD,MRG_MYISAM,OPTIMIZE," +
        "OPTIONALLY,OUTFILE,PURGE,REGEXP,RENAME,REPLACE,REQUIRE,RETURNS," +
        "RLIKE,RTREE,SHOW,SONAME,SPATIAL,SQL_BIG_RESULT,SQL_CALC_FOUND_ROWS," +
        "SQL_SMALL_RESULT,SSL,STARTING,STRAIGHT_JOIN,STRIPED,TABLES," +
        "TERMINATED,TINYBLOB,TINYINT,TINYTEXT,TYPES,UNLOCK,UNSIGNED,USE," +
        "USER_RESOURCES,VARBINARY,VARCHARACTER,WARNINGS,XOR,YEAR_MONTH," +
        "ZEROFILL";

    /**
     * Constructor.
     * Overridden so we can add on our own list of NON SQL92 reserved words
     * which is returned incorrectly with the JDBC driver.
     * @param metadata MetaData for the DB
     **/
    public MySQLAdapter(DatabaseMetaData metadata)
    {
        super(metadata);

        reservedKeywords.addAll(parseKeywordList(NONSQL92_RESERVED_WORDS));
    }

    /**
     * An alias for this adapter.
     * @return The alias
     */
    public String getVendorID()
    {
        return "mysql";
    }

    /**
```

```
     * MySQL, when using AUTO_INCREMENT, requires the primary key specified
     * in the CREATE TABLE, so we do nothing here.
     *
     * @param pkName The name of the primary key to add.
     * @param pk An object describing the primary key.
     * @return The statement to add the primary key separately
     */
    public String getAddPrimaryKeyStatement(SQLIdentifier pkName, PrimaryKey pk)
    {
        return null;
    }

    /**
     * Whether the datastore supports specification of the primary key in
     * CREATE TABLE statements.
     * @return Whetehr it allows "PRIMARY KEY ..."
     */
    public boolean supportsPrimaryKeyInCreateStatements()
    {
        return true;
    }

    /**
     * Method to return the CREATE TABLE statement.
     * Versions before 5 need INNODB table type selecting for them.
     * @param table The table
     * @param columns The columns in the table
     * @return The creation statement
     **/
    public String getCreateTableStatement(TableImpl table, Column[] columns)
    {
        StringBuffer createStmt = new StringBuffer(super.getCreateTableStatement(
table,columns));

        // Versions before 5.0 need InnoDB table type
        if (datastoreMajorVersion < 5)
        {
            createStmt.append(" TYPE=INNODB");
        }

        return createStmt.toString();
    }

    ...
}
```

So here we've shown a snippet from the MySQL DatastoreAdapter. We basically take much behaviour from the base class but override what we need to change for our RDBMS. You should get the idea by now. Just go through the Javadocs of the superclass and see what you need to override.

A final step that is optional here is to integrate your new adapter as a DataNucleus plugin. To do this you need to package it with a file `plugin.xml`, specified at the root of the CLASSPATH, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="MyCompany DataNucleus plug-in" provider-name="MyCompany">
    <extension point="org.datanucleus.store.rdbms.datastoreadapter">
        <datastore-adapter vendor-id="myname" class-name="mydomain.MyDatastoreAdapter"
priority="10"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

Where the *myname* specified is a string that is part of the JDBC "product name" (returned by "DatabaseMetaData.getDatabaseProductName()"). If there are multiple adapters for the same *vendor-id* defined, the attribute *priority* is used to determine which one is used. The adapter with the highest number is chosen. Note that the behaviour is undefined when two or more adapters with *vendor-id* have the same priority. All adapters defined in DataNucleus and its official plugins use priority values between *0* and *9*. So, to make sure your adapter is chosen, use a value higher than that.

7.4. RDBMS Connection Pooling

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the pooling of connections to RDBMS datastores. DataNucleus provides a large selection of connection pools (DBCP, C3P0, HikariCP, BoneCP) but is structured so that you can easily add your own variant and have it usable within your DataNucleus usage.

DataNucleus requires a DataSource to define the datastore in use and consequently allows use of connection pooling. DataNucleus provides plugins for various different pooling products, shown below. You can easily define your own plugin for pooling. You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.store.rdbms.connectionpool**.

Plugin extension-point	Key	Description	Location
org.datanucleus.store.rdbms.connectionpool	dbcp-builtin	RDBMS connection pool, using Apache DBCP builtin	datanucleus-rdbms
org.datanucleus.store.rdbms.connectionpool	bonecp	RDBMS connection pool, using BoneCP	datanucleus-rdbms
org.datanucleus.store.rdbms.connectionpool	c3p0	RDBMS connection pool, using C3P0	datanucleus-rdbms
org.datanucleus.store.rdbms.connectionpool	dbcp	RDBMS connection pool, using Apache DBCP	datanucleus-rdbms
org.datanucleus.store.rdbms.connectionpool	tomcat	RDBMS connection pool, using Tomcat pool	datanucleus-rdbms

The following sections describe how to create your own connection pooling plugin for DataNucleus.

7.4.1. Interface

If you have your own DataSource connection pooling implementation you can easily use it with DataNucleus. Javadoc. DataNucleus defines a ConnectionPoolFactory interface and you need to implement this.

```
package org.datanucleus.store.rdbms.connectionpool;

public interface ConnectionPoolFactory
{
    /**
     * Method to make a ConnectionPool for use within DataNucleus.
     * @param storeMgr StoreManager
     * @return The ConnectionPool
     * @throws Exception Thrown if an error occurs during creation
     */
    public ConnectionPool createConnectionPool(StoreManager storeMgr);
}
```

where you also define a ConnectionPool Javadoc.

```
package org.datanucleus.store.rdbms.connectionpool;

public interface ConnectionPool
{
    /**
     * Method to call when closing the StoreManager down, and consequently to close
the pool.
     */
    void close();

    /**
     * Accessor for the pooled DataSource.
     * @return The DataSource
     */
    DataSource getDataSource();
}
```

7.4.2. Plugin Specification

The only thing required now is to register this plugin with DataNucleus when you start up your application. To do this create a file `plugin.xml` and put it in your JAR at the root of the CLASSPATH, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.connectionpool">
        <connectionpool-factory name="mypool" class-
name="mydomain.MyConnectionPoolFactory"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

7.4.3. Plugin Usage

The only thing remaining is to use your new *ConnectionPoolFactory* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the persistence property **datanucleus.connectionPoolingType** to *mypool* (the "name" you specified in the `plugin.xml` file).

7.5. RDBMS Identifier Factory

DataNucleus is developed as a plugin-driven framework and one of the components that is pluggable is the naming of datastore identifiers. DataNucleus provides its own internal identifier factories, but also allows you to plugin your own factory. Identifiers are required when using an RDBMS datastore to define the name of components in the datastore, such as table/columns.

DataNucleus provides the mechanism to generate datastore identifiers (table/column names) when none are defined by the users metadata/annotations. In addition to the default JDO factory there is also an identifier factory that generates identifiers consistent with the JPA1 specification. You can extend DataNucleus's capabilities using the plugin extension **org.datanucleus.store.rdbms.identifierfactory**.

Plugin extension-point	Key	Description	Location
org.datanucleus.store.rdbms.identifierfactory	jpox	Identifier Factory providing DataNucleus JPOX default namings	datanucleus-core
org.datanucleus.store.rdbms.identifierfactory	datanucleus1	Identifier Factory providing DataNucleus 1.x default namings	datanucleus-core
org.datanucleus.store.rdbms.identifierfactory	datanucleus2	Identifier Factory providing DataNucleus 2.x+ namings	datanucleus-core
org.datanucleus.store.rdbms.identifierfactory	jpa	Identifier Factory providing JPA-compliant namings	datanucleus-core

7.5.1. Interface

Any identifier factory plugin will need to implement *org.datanucleus.store.rdbms.identifier.IdentifierFactory* Javadoc. So you need to implement the

following interface

```
package org.datanucleus.store.rdbms.identifier;

public interface IdentifierFactory
{
    /**
     * Accessor for the datastore adapter that we are creating identifiers for.
     * @return The datastore adapter
     */
    DatastoreAdapter getDatastoreAdapter();

    /**
     * Accessor for the identifier case being used.
     * @return The identifier case
     */
    IdentifierCase getIdentifierCase();

    /**
     * Accessor for an identifier for use in the datastore adapter
     * @param identifier The identifier name
     * @return Identifier name for use with the datastore adapter
     */
    String getIdentifierInAdapterCase(String identifier);

    /**
     * To be called when we want an identifier name creating based on the
     * identifier. Creates identifier for COLUMN, FOREIGN KEY, INDEX and TABLE
     * @param identifierType the type of identifier to be created
     * @param identifierName The identifier name
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newIdentifier(IdentifierType identifierType, String
identifierName);

    /**
     * Method to use to generate an identifier for a datastore field with the supplied
name.
     * The passed name will not be changed (other than in its case) although it may
     * be truncated to fit the maximum length permitted for a datastore field
identifier.
     * @param identifierName The identifier name
     * @return The DatastoreIdentifier for the table
     */
    DatastoreIdentifier newDatastoreContainerIdentifier(String identifierName);

    /**
     * Method to return a Table identifier for the specified class.
     * @param md Meta data for the class
     * @return The identifier for the table
```

```
     */
    DatastoreIdentifier newDatastoreContainerIdentifier(AbstractClassMetaData md);

    /**
     * Method to return a Table identifier for the specified field.
     * @param fmd Meta data for the field
     * @return The identifier for the table
     */
    DatastoreIdentifier newDatastoreContainerIdentifier(AbstractMemberMetaData fmd);

    /**
     * Method to use to generate an identifier for a datastore field with the supplied
name.
     * The passed name will not be changed (other than in its case) although it may
     * be truncated to fit the maximum length permitted for a datastore field
identifier.
     * @param identifierName The identifier name
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newDatastoreFieldIdentifier(String identifierName);

    /**
     * Method to create an identifier for a datastore field where we want the
     * name based on the supplied java name, and the field has a particular
     * role (and so could have its naming set according to the role).
     * @param javaName The java field name
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g FK, Index ?
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newDatastoreFieldIdentifier(String javaName, boolean embedded,
int fieldRole);

    /**
     * Method to generate an identifier name for reference field, based on the
metadata for the
     * field, and the ClassMetaData for the implementation.
     * @param refMetaData the MetaData for the reference field
     * @param implMetaData the AbstractClassMetaData for this implementation
     * @param implIdentifier PK identifier for the implementation
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g FK, collection
element ?
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newReferenceFieldIdentifier(AbstractMemberMetaData
refMetaData,
        AbstractClassMetaData implMetaData, DatastoreIdentifier implIdentifier,
        boolean embedded, int fieldRole);

    /**
```

```
     * Method to return an identifier for a discriminator datastore field.
     * @return The discriminator datastore field identifier
     */
    DatastoreIdentifier newDiscriminatorFieldIdentifier();

    /**
     * Method to return an identifier for a version datastore field.
     * @return The version datastore field identifier
     */
    DatastoreIdentifier newVersionFieldIdentifier();

    /**
     * Method to return a new Identifier based on the passed identifier, but adding on
the passed suffix
     * @param identifier The current identifier
     * @param suffix The suffix
     * @return The new identifier
     */
    DatastoreIdentifier newIdentifier(DatastoreIdentifier identifier, String suffix);

    // RDBMS types of identifiers

    /**
     * Method to generate a join-table identifier. The identifier could be for a
foreign-key
     * to another table (if the destinationId is provided), or could be for a simple
column
     * in the join table.
     * @param ownerFmd MetaData for the owner field
     * @param relatedFmd MetaData for the related field (if bidirectional)
     * @param destinationId Identifier for the identity field of the destination table
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g FK, collection
element ?
     * @return The identifier.
     */
    DatastoreIdentifier newJoinTableFieldIdentifier(AbstractMemberMetaData ownerFmd,
        AbstractMemberMetaData relatedFmd,
        DatastoreIdentifier destinationId, boolean embedded, int fieldRole);

    /**
     * Method to generate a FK/FK-index field identifier.
     * The identifier could be for the FK field itself, or for a related index for the
FK.
     * @param ownerFmd MetaData for the owner field
     * @param relatedFmd MetaData for the related field (if bidirectional)
     * @param destinationId Identifier for the identity field of the destination table
(if strict FK)
     * @param embedded Whether the identifier is for a field embedded
     * @param fieldRole The role to be performed by this column e.g owner, index ?
     * @return The identifier
```

```
     */
    DatastoreIdentifier newForeignKeyFieldIdentifier(AbstractMemberMetaData ownerFmd,
        AbstractMemberMetaData relatedFmd,
        DatastoreIdentifier destinationId, boolean embedded, int fieldRole);

    /**
     * Method to return an identifier for an index (ordering) datastore field.
     * @param mmd Metadata for the field/property that we require to add an
index(order) column for
     * @return The index datastore field identifier
     */
    DatastoreIdentifier newIndexFieldIdentifier(AbstractMemberMetaData mmd);

    /**
     * Method to return an identifier for an adapter index datastore field.
     * An "adapter index" is a column added to be part of a primary key when some
other
     * column cant perform that role.
     * @return The index datastore field identifier
     */
    DatastoreIdentifier newAdapterIndexFieldIdentifier();

    /**
     * Method to generate an identifier for a sequence using the passed name.
     * @param sequenceName the name of the sequence to use
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newSequenceIdentifier(String sequenceName);

    /**
     * Method to generate an identifier for a primary key.
     * @param table the table
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newPrimaryKeyIdentifier(DatastoreContainerObject table);

    /**
     * Method to generate an identifier for an index.
     * @param table the table
     * @param isUnique if the index is unique
     * @param seq the sequential number
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newIndexIdentifier(DatastoreContainerObject table,
                boolean isUnique, int seq);

    /**
     * Method to generate an identifier for a candidate key.
     * @param table the table
     * @param seq Sequence number
     * @return The DatastoreIdentifier
```

```
     */
    DatastoreIdentifier newCandidateKeyIdentifier(DatastoreContainerObject table, int
seq);

    /**
     * Method to create an identifier for a foreign key.
     * @param table the table
     * @param seq the sequential number
     * @return The DatastoreIdentifier
     */
    DatastoreIdentifier newForeignKeyIdentifier(DatastoreContainerObject table, int
seq);
}
```

Be aware that you can extend *org.datanucleus.store.rdbms.identifier.AbstractIdentifierFactory* Javadoc.

7.5.2. Implementation

Let's assume that you want to provide your own identifier factory *MyIdentifierFactory*

```
package mydomain;

import org.datanucleus.store.rdbms.identifier.AbstractSIdentifierFactory

public class MyIdentifierFactory extends AbstractIdentifierFactory
{
    /**
     * Constructor.
     * @param dba Datastore adapter
     * @param clr ClassLoader resolver
     * @param props Map of properties with String keys
     */
    public MyIdentifierFactory(DatastoreAdapter dba, ClassLoaderResolver clr, Map
props)
    {
        super(dba, clr, props);
        ...
    }

    .. (implement the rest of the interface)
}
```

7.5.3. Plugin Specification

When we have defined our "IdentifierFactory" we just need to make it into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.identifierfactory">
        <identifierfactory name="myfactory" class-name=
"mydomain.MyIdentifierFactory"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

7.5.4. Plugin Usage

The only thing remaining is to use your new *IdentifierFactory* plugin. You do this by having your plugin in the CLASSPATH at runtime, and setting the PMF property *datanucleus.identifierFactory* to *myfactory* (the name you specified in the `plugin.xml` file).

7.6. RDBMS SQL Methods

DataNucleus supports specific JDOQL methods / JPQL functions for RDBMS datastores. This support is extensible. DataNucleus provides support for the majority of SQL methods that you are ever likely to need but is structured so that you could add on support for your own easily enough. The following sections describe how to create your own SQL Method plugin for DataNucleus.

7.6.1. Interface

Any SQL Method plugin will need to implement *org.datanucleus.store.rdbms.sql.method.SQLMethod* Javadoc. So you need to implement the following interface

```
import org.datanucleus.store.rdbms.sql.method;

public interface SQLMethod
{
    /**
     * Return the expression for this SQL function.
     * @param stmt The SQL Statement
     * @param expr The expression that it is invoked on
     * @param args Arguments passed in
     * @return The SQL expression using the SQL function
     */
    public SQLExpression getExpression(SQLStatement stmt, SQLExpression expr, List
<SQLExpression> args);
}
```

7.6.2. Implementation

So there is only one method to provide in your implementation. The arguments to this are

- The expression on which the method is invoked. So if you have *{string}.myMethod(args)* then the first argument will be a *StringExpression*. If the method is a static function then this argument is null
- The args are the arguments passed in to the method call. They will be SQLExpression/SQLLiteral

So if we wanted to support *{String}.length()* as an example, so we define our class as

```
package mydomain;

import java.util.List;
import java.util.ArrayList;

import org.datanucleus.exceptions.NucleusException;
import org.datanucleus.store.rdbms.sql.expression.NumericExpression;
import org.datanucleus.store.rdbms.sql.expression.SQLExpression;
import org.datanucleus.store.rdbms.sql.expression.StringExpression;

public class MyStringLengthMethod extends AbstractSQLMethod
{
    public SQLExpression getExpression(SQLStatement stmt, SQLExpression expr, List
args)
    {
        if (expr instanceof StringExpression)
        {
            ArrayList funcArgs = new ArrayList();
            funcArgs.add(expr);
            return new NumericExpression("CHAR_LENGTH", funcArgs);
        }
        else
        {
            throw new NucleusException(Localiser.msg("060001", "length", expr));
        }
    }
}
```

So in this implementation when the user includes *{string}.length()* this is translated into the SQL *CHAR_LENGTH({string})* which will certainly work on some RDBMS. Obviously you could use this extension mechanism to support a different underlying SQL function.

7.6.3. Plugin Specification

So we now have our custom SQL method and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.sql_method">
        <sql-method class="java.lang.String" method="length" datastore="h2"
            evaluator="mydomain.MyStringLengthMethod"/>
    </extension>
</plugin>
```

If implementing support for a method that is static (e.g JDOHelper.getObjectId()) omit the "class" argument from the plugin.xml entry, and put the method as "JDOHelper.getObjectId" Note that you also require a `MANIFEST.MF` file as described above.

So we defined calls to a method *length* for the type *java.lang.String* for the datastore "h2" to use our evaluator. Simple! Whenever this method is encountered in a query from then on for the H2 database it will use our method evaluator.

7.7. RDBMS SQL Expression Support

Whilst processing JDOQL/JPQL queries, DataNucleus internally uses "expressions" for java types. When you add support for a new java type, if you want to query fields of that type you will need to provide this.

While you can define details of how your type is queryable, we only document here what you would do for a type that is mapped to a String (since that's the most common use-case). Here you simply need to add an entry into `plugin.xml` defining which expression and literal class you would use for handling that type.

7.7.1. Plugin Specification

Say we have a java type that is mapped using *mydomain.MyTypeMapping*. Simply add a file `plugin.xml` to your JAR at the root like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.sql_expression">
        <sql-expression mapping-class="mydomain.MyTypeMapping"
            literal-class="org.datanucleus.store.rdbms.sql.expression.StringLiteral"
            expression-
class="org.datanucleus.store.rdbms.sql.expression.StringExpression"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

7.8. RDBMS SQL Operations

Whilst processing JDOQL/JPQL queries, DataNucleus internally uses "operations" between particular java types. DataNucleus provides support for all major operations typically handling them internally but occasionally handing off to an SQL function where one is more appropriate. However the codebase is structured so that you could add on support for your own easily enough

The following sections describe how to create your own SQL Operation plugin for DataNucleus.

7.8.1. Interface

Any SQL operation plugin will need to implement *org.datanucleus.store.rdbms.sql.operation.SQLOperation* Javadoc. So you need to implement the following interface

```
import org.datanucleus.store.rdbms.sql.method;

public interface SQLOperation
{
    /**
     * Return the expression for this SQL function.
     * @param expr Left hand expression
     * @param expr2 Right hand expression
     * @return The SQL expression for the operation
     */
    public SQLExpression getExpression(SQLExpression expr, SQLExpression expr2);
}
```

7.8.2. Implementation

So there is only one method to provide in your implementation. The arguments to this are

- The expression on the left hand side of the operation. So if you have *{expr1} {operation} {expr2}* then the first argument will be *{expr1}*
- The expression on the right hand side of the operation. So if you have *{expr1} {operation} {expr2}* then the first argument will be *{expr2}*

So if we wanted to support *modulus (%)* (so something like *expr1 % expr2*) and wanted to use the SQL function "MOD" to provide this then we define our class as

```
package mydomain;

import java.util.ArrayList;

import org.datanucleus.exceptions.NucleusException;
import org.datanucleus.store.rdbms.sql.operation.SQLOperation;
import org.datanucleus.store.rdbms.sql.expression.SQLExpression;
import org.datanucleus.store.rdbms.sql.expression.NumericExpression;

public class MyModOperation implements SQLOperation
{
    public SQLExpression getExpression(SQLExpression expr, SQLExpression expr2)
    {
        ArrayList args = new ArrayList();
        args.add(expr);
        args.add(expr2);
        return new NumericExpression("MOD", args);
    }
}
```

So in this implementation when the user includes *{expr1} % {expr2}* this is translated into the SQL *MOD({expr1}, {expr2})* which will certainly work on some RDBMS. Obviously you could use this extension mechanism to support a different underlying SQL function.

7.8.3. Plugin Specification

So we now have our custom SQL method and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root, like this

```
    <?xml version="1.0"?>
    <plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
        <extension point="org.datanucleus.store.rdbms.sql_operation">
            <sql-operation name="mod" datastore="hsql"
                evaluator="mydomain.MyModOperation"/>
        </extension>
    </plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

So we defined calls to an operation *mod* for the datastore "hsql" to use our evaluator. Simple! Whenever this operation is encountered in a query from then on for the HSQL database it will use our operation evaluator.

7.9. RDBMS SQL Table Namer

Whilst generating SQL, DataNucleus allows an amount of control over the aliases used for tables in that SQL. DataNucleus provides a few options out of the box. The default is *alpha-scheme* which

names tables based on the table-group they are in and the number in that group, so giving names like A0, A1, A2, B0, B1, C0, D0. It also provides a simpler option t-scheme that names all tables as "T{number}" so T0, T1, T2, T3, etc.

The following sections describe how to create your own SQL Table Namer plugin for DataNucleus.

7.9.1. Interface

Any SQL Table Namer plugin will need to implement *org.datanucleus.store.rdbms.sql.SQLTableNamer* Javadoc. So you need to implement the following interface

```
package org.datanucleus.store.rdbms.sql;

import org.datanucleus.store.rdbms.sql.SQLStatement;
import org.datanucleus.store.mapped.DatastoreContainerObject;

public interface SQLTableNamer
{
    /**
     * Method to return the alias to use for the specified table.
     * @param stmt The statement where we will use the table
     * @param table The table
     * @return The alias to use
     */
    public String getAliasForTable(SQLStatement stmt, DatastoreContainerObject table);
}
```

7.9.2. Implementation

So there is only one method to provide in your implementation. The arguments to this are

- The SQLStatement that is being constructed and that we create the names for
- The table that we want to generate the alias for

Lets just go through our default namer scheme to understand how it works

```
package mydomain;

import org.datanucleus.store.rdbms.sql.SQLTableNamer;
import org.datanucleus.store.rdbms.sql.SQLStatement;
import org.datanucleus.store.mapped.DatastoreContainerObject;

public class MySQLTableNamer implements SQLTableNamer
{
    public String getAliasForTable(SQLStatement stmt, DatastoreContainerObject table)
    {
        if (stmt.getPrimaryTable() == null)
        {
            return "T0";
        }
        else
        {
            return "T" + (stmt.getNumberOfTables() < 0 ? "1" : (stmt.
getNumberOfTables()+1));
        }
    }
}
```

So we simply name the primary table of the statement as "T0", and then all subsequent tables based on the number of the table. That was hard!

7.9.3. Plugin Specification

So we now have our custom SQL method and we just need to make this into a DataNucleus plugin. To do this you simply add a file `plugin.xml` to your JAR at the root, like this

```
<?xml version="1.0"?>
<plugin id="mydomain" name="DataNucleus plug-ins" provider-name="My Company">
    <extension point="org.datanucleus.store.rdbms.sql_tablenamer">
        <sql-tablenamer name="my-t-scheme" class="mydomain.MySQLTableNamer"/>
    </extension>
</plugin>
```

Note that you also require a `MANIFEST.MF` file as described above.

So now if we define a query using the extension *datanucleus.sqlTableNamingStrategy* set to "my-t-scheme" then it will use our table namer.