Java LDAP Persistence with DataNucleus

by Stefan Seelmann

1 Motivation

When a Java developer needs to access a directory server over LDAP he has the choice between multiple APIs.

There are low-level APIs which provide direct access to the operations defined by the LDAP protocol, including controls and extended operations. Some mature implementations are the Netscape LDAP SDK and JLDAP (Novell/OpenLDAP). There are also some newer implementations from UnboundID and ongoing implementations from Apache Directory and OpenDS projects, which leverage improvements of the Java language (Generics, NIO).

Then there is JNDI (Java Naming and Directory Interface). It is included in Java SE and thus widely used. The problem with JNDI is that it abstracts the LDAP protocol. It defines its own wording: The *bind()* method for example creates a new entry instead of doing an authentication. Spring-LDAP can be used to simplify JNDI and to avoid boiler-plate code.

One drawback of all those APIs is that a Java developer has to deal with the LDAP protocol. That makes sense when doing things like authentication or when using a specific control, but not when the developer only wants to work with the objects stored in the directory.

In the Java universe two major persistence standards exist: JPA and JDO. While JPA (Java Persistence API) was designed for RDBMS only, JDO (Java Data Objects) was designed independent of the underlying datastore.

This paper will discuss the usage of JDO and its implementation DataNucleus for LDAP persistence.

2 JDO and DataNucleus Overview

The aim of JDO is to provide a Java-centric API to access persistent data. DataNucleus is the reference implementation of the JDO API. It also supports other APIs (JPA, REST) and supports a wide range of datastores (RDBMS, db4o, LDAP, Excel, XML, NeoDatis, JSON, ODF, Google BigTable, Hadoop HBase).

The JDO specification [1] consists of three main parts:

- Persistence Definition
- Persistence API
- Query Language

2.1 Persistence Definition

The persistence metadata describes what data to persist and where to persist it in the datastore. This can be done by adding annotations to the persistable domain classes or by specifying XML metadata. Best practise is to use a combination of both: specifying basic persistence info as annotations, and adding datastore specific info to XML files.

```
                          @PersistenceCapable
public class User         public class User              <class name="User">
{                         {
                            @Persistent(primaryKey="true")   <field name="uid" primaryKey="true" />
  String uid;               String uid;
                            @Persistent                      <field name="phoneNumbers"
  Set<String> phoneNumbers; Set<String> phoneNumbers;          persistence-modifier="persistent"/>
    ...                       ...
}                         }                             </class>
```

Each persistable class must implement the `javax.jdo.spi.PersistenceCapable`. DataNucleus uses a byte code enhancer to add this interface and its methods to Java classes.

2.2 Persistence API

The central class of the persistence API is the `javax.jdo.PersistenceManager` class. It is obtained from the `javax.jdo.PersistenceManagerFactory`.

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

For storing and loading objects, the persistence manager provides methods like `makePersistent()`, `deletePersistent()`, and `getObjectById()`.

2.3 Query API and Language

The query API and language is used to retrieve objects. A query could either be constructed using the `javax.jdo.Query` or by a SQL-like string.

A query may contain elements like the candidate class to search for, a filter, an ordering statement, and aggregation instructions. A query is translated for the native datastore. If the datastore can't handle such a query, DataNucleus provides an in-memory evaluation of queries.

Another important feature of JDO are fetch groups. They are used to control which fields of an object should be loaded from the datastore.

3 DataNucleus LDAP Store Basic Example

The DataNucleus LDAP store [4] is a plugin in DataNucleus that supports persisting objects to directory server using the LDAP protocol. To show how the LDAP store works a basic example is used. The LDAP server contains user entries with object class *inetOrgPerson* below *ou=Users,dc=example,dc=com*.

3.1 Domain Class and Persistence Metadata

The obvious approach is to map one entry to one Java object. So the first step is to create a domain class `User` and to define the persistence metadata.

```
 1  @PersistenceCapable(table = "ou=Users,dc=example,dc=com",
 2      schema = "top,person,organizationalPerson,inetOrgPerson")
 3  public class User
 4  {
 5      @Persistent(column = "cn", primaryKey = "true")
 6      private String fullName;
 7
 8      @Persistent(column = "givenName")
 9      private String firstName;
10
11      @Persistent(column = "sn")
12      private String lastName;
13
14      @Persistent(column = "employeeNumber")
15      private long personNumber;
16
17      @Persistent(column = "description", defaultFetchGroup = "true")
18      private Calendar dayOfBirth;
19
20      @Persistent(column = "telephoneNumber", defaultFetchGroup = "true")
21      private Set<String> phoneNumbers = new HashSet<String>();
22      ...
23  }
```

The class is a simple Java Bean class, the constructor as well as getters and setters are omitted. This example uses annotations only to define persistence metadata. Note: The JDO specification defines common annotations and additional ORM (Object Relational Mapping) specific annotations. For the LDAP store some of these ORM annotations are reused. JDO also allows to define custom annotations, however their extensive usage would blow up the source code.

The class is marked to be persisted with the `@PersistenceCapable` (lines 1 and 2). The `table` parameter defines the container entry where objects of type `User` are persisted. The `schema` parameter defines the object classes of the entry. This information is also used to lookup `User` objects: the `table` parameter is used as search base, the default search scope is one level and the object classes are used to construct the search filter.

The fields that should be persisted are marked with the `@Persistent` annotation. The `column` parameter is used to define which LDAP attribute to use. On line 5 an additional parameter `primaryKey="true"` is used to mark the RDN attribute. Beside Strings and primitive types also complex types can be persisted. On lines 17 and 18 an `java.util.Calendar` field is persisted. Date and Calendar fields are automatically converted to generalized time syntax. An additional parameter `defaultFetchGroup="true"` must be specified to load this field automatically when

retrieving the object. Lines 20 and 21 define a set with multiple phone numbers. Collections of String or wrappers of primitives are automatically mapped to multi-valued attributes.

3.2 Persistence Code

Now the persitence code can be writen.

```
 1  public class Main
 2  {
 3      public static void main( String[] args )
 4      {
 5          // obtain the persistence manager
 6          PersistenceManagerFactory pmf =
 7                JDOHelper.getPersistenceManagerFactory( "datanucleus.properties" );
 8          PersistenceManager pm = pmf.getPersistenceManager();
 9
10          // create a user
11          User user = new User( "Bugs Bunny", "Bugs", "Bunny" );
12          user = pm.makePersistent( user );
13
14          // search for users which last name begins with 'B'
15          Query query = pm.newQuery( User.class );
16          query.setFilter( "lastName.startsWith('B')" );
17          Collection<User> users = ( Collection<User> ) query.execute();
18          System.out.println( users );
19
20          // delete the user
21          pm.deletePersistent( user );
22      }
23  }
```

First the persistence manager is obtained (lines 6-8). In lines 10 and 11 a `User` object is created and persisted to the directory. Lines 14-16 demonstrate how to query for `User` objects. At last at line 20 the object is deleted from directory.

3.3 Run

To run the example the DataNucleus libraries and dependencies are needed, they can be downloaded from [2]. There is a separate archive for the LDAP store called datanucleus-accessplatform-ldap-<version>.zip. Additional the domain class must be enhanced, which could be a tricky part. The DataNucleus website [3] shows all available enhancement options. A recommended way to manage dependencies and to build a project is to use a build tool like Maven or Ant.

A final preparation is the configuration of the persistence manager. A basic persistence configuration *datanucleus.properties* for LDAP looks like this:

```
1  javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
2  javax.jdo.option.ConnectionDriverName=com.sun.jndi.ldap.LdapCtxFactory
3  javax.jdo.option.ConnectionURL=ldap://localhost:10389
4  javax.jdo.option.ConnectionUserName=uid=admin,ou=system
5  javax.jdo.option.ConnectionPassword=secret
```

Line 1 specified the persistence manager factory, here the DataNucleus implementation is used. The LDAP store uses JNDI, line 2 defines the context factory. Line 3 define the LDAP server's host and port. Line 4 and 5 specify the authentication credentials.

4 DataNucleus LDAP Store Details

This chapter describes features and design decisions of the LDAP datastore.

4.1 Basic Mapping

In general, each object of a persistence capable object is mapped to its own LDAP entry. The class must be marked with the `@PersistenceCapable` annotation. The `schema` parameter of the annotations defines the object classes for the entry. When persisting an object the object classes are automatically added, the Java developer doesn't have to deal with it.

The fields of the Java objects are mapped to attributes of the LDAP entries. Strings and primitive fields are automatically persisted, other fields must be marked with the `@Persistent` annotation. The `column` parameter is used to define the LDAP attribute name, if omitted the field name is used.

String, primtives, wrappers of primitives, BigDecimal, BigInteger, Date and Calendar fields are stored as single-valued attribute. Date and Calendar are converted to generalized time syntax. Collections, Sets and Lists of these data types are stored as multi-valued attributes. If lists are marked with the `@Order` annotation the values are prefixed with an index {i}. For other non-persistable Java types that have a string representation it is possible to create own `ObjectStringConverter`s. Byte array fields are stored as binary attribute to the entry.

4.2 Object Identity and Distinguished Name

JDO requires that each persistable object has an identity. The LDAP store only supports application identity and single-field identity. This means that exactly one field of the object is used as primary key and must be marked as primary key (`@Persistent(primaryKey="true")` or `@PrimaryKey`). This field becomes the RDN attribute of the entry. As a consequence only single-valued RDNs are supported. All single-valued non-persistable field types described in the previous section may be used as primary key field.

The parent entry of an entry in the DIT is controlled via the `table` parameter. The syntax of the parameter value can be:

- distinguished name: Defines the container entry for that type. All object of that type are persisted as child entries below this container entry. To retrieve objects of that type this DN is used as search base (plus one-level scope and a filter constructed from `schema` parameter). It is possible to store different types into the same container, as Active Directory does with User and Group entries.
- LDAP URL (only dn, scope and filter are used: ldap:///<dn>??<scope>?<filter>). This allows a more fine-grained definition of the entry location. The DN has the same meaning as above. Scope *base* means that there is only one single instance of that class it is persisted at the given DN. Scope *sub* makes only sense with hierarchical mapping, see below. The filter could be used as a discriminator for types that have the same object classes but vary in some other attribute.

- {<parent field name>}: A reference to the parent object field, used for hierarchical relationships. The child Java objects need a non-null reference to its logical parent. Such a child object is then persisted a child entry of the parent entry. This could also be recursive (the parent is itself a child of its parent).

4.3 Relationship Mapping Strategies

DataNucleus implements different strategies for persisting relationships that cover LDAP best practices.

4.3.1 By Distinguished Name

One entry points to another entry by storing the distinguished name of the target entry. This is used by group entries using the groupOfNames object class. This strategy is used by default for fields referencing persistence capable objects.

```
 1  @PersistenceCapable(table = "ou=Groups,dc=example,dc=com", schema = "top,groupOfNames")
 2  public class Group
 3  {
 4      @Persistent(column = "member")
 5      @Extension(vendorName = "datanucleus", key = "empty-value", value = "uid=admin,ou=system")
 6      protected Set<User> members = new HashSet<User>();
 7      ...
 8  }
 9
10  @PersistenceCapable(table = "ou=Users,dc=example,dc=com",
11      schema = "top,person,organizationalPerson,inetOrgPerson"
12  public class User
13  {
14      @Persistent(mappedBy = "members")
15      protected Set<Group> memberOf = new HashSet<Group>();
16      ...
17  }
```

Here we see an N:M bidirectional relationship. The `Group` class is the owner of the relationship (mapped-by specified on the `User` side of the relation) and defines the attribute `member` where DNs are stored (line 5). The extension in line 6 covers the case that the group contains no member, in this case a dummy value is added because member is a mandatory attribute of groupOfNames. The field `memberOf` of the `User` class isn't stored in the directory, hence it is marked as mapped by the other field.

4.3.2 By Attribute Match

This is similar to the previous strategy, however only an attribute value of the target entry is stored. This is used by posixGroup and posixAccount object classes. The only difference is to add an `@Join(column="gidNumber")` annotation. The `Join` tells DataNucleus to use attribute matching, the `column` specifies the attribute of the target entry that contains the value to match.

4.3.3 Hierarchical with Parent Reference

The main challenge with hierarchical mapping is that the distinguished name (DN) of children depends on the DN of their parent. Each child object needs to have a reference to its parent object, to be able to determine where it should be persisted in the DIT. The child class metadata doesn't contain a fixed container DN but the name of the field holding the reference to the parent object.

The following example demonstrates this strategy. All `Department` objects are persisted into *dc=example,dc=com*. The `User` objects are persisted below their department entries.

```
@PersistenceCapable(table="dc=example,dc=com",
    schema="top,organizationalUnit")
public class Department
{
    ...
}

@PersistenceCapable(table="{department}",
    schema="top,person,organizationalPerson,inetOrgPerson")
public class User
{
    ...
    @Persistent(defaultFetchGroup = "true")
    private Department department;
    ...
}
```

```
dc=example,dc=com
|
|-- ou=Sales
|   |-- cn=Bugs Bunny
|   |-- cn=Daffy Duck
|   |-- ...
|
|-- ou=Engineering
|   |-- cn=Speedy Gonzales
|   |-- ...
|
|-- ...
```

4.3.4 Embedded as Child Entry

This strategy is used to design a containment (is-part-of) relationship. Only the owner object has its own identity. The embedded objects can't be retrieved on their own, they could only be retrieved by traversing the object graph starting at the owner object. If the owner object is loaded all the embedded objects are loaded too.

```
public class Person
{
    private String fullName;
    private String firstName;
    private String lastName;
    private Account account;
    ...
}

public class Account
{
    private String uid;
    private String password;
    ...
}
```

```
dn: cn=Bugs Bunny,ou=Persons,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Bugs Bunny
givenName: Bugs
sn: Bunny

dn: uid=bbunny,cn=Bugs Bunny,ou=Persons,dc=example,dc=com
objectClass: top
objectClass: account
objectClass: simpleSecurityObject
uid: bbunny
userPassword: secret
```

The JDO metadata for this kind of mapping looks like this. The `Account` class is marked as embedded-only and no DN is specified.

```
<class name="Person" table="ou=Persons,dc=example,dc=com"
       schema="top,person,organizationalPerson,inetOrgPerson">
    <field name="fullName" primary-key="true" column="cn" />
    <field name="firstName" column="givenName" />
    <field name="lastName" column="sn" />
    <field name="account">
        <embedded />
    </field>
</class>
<class name="Account" embedded-only="true" schema="top,account,simpleSecurityObject">
    <field name="uid" primary-key="true" column="uid" />
    <field name="password" column="userPassword" />
</class>
```

4.3.5 Embedded into Owner Entry

This is similar to the previous strategy, however the fields of the embedded object are stored within the owner entry. As an example personal information of a person and its account data could be stored in one inetOrgPerson entry.

```
public class Person                     dn: cn=Bugs Bunny,ou=Persons,dc=example,dc=com
{                                       objectClass: top
    private String fullName;            objectClass: person
    private String firstName;           objectClass: organizationalPerson
    private String lastName;            objectClass: inetOrgPerson
    private Account account;            cn: Bugs Bunny
    ...                                 givenName: Bugs
}                                       sn: Bunny
public class Account                    uid: bbunny
{                                       userPassword: secret
    private String uid;
    private String password;
    ...
}
```

The JDO metadata for this kind of mapping looks like this, there is no separate class definition for the `Account` class:

```
<class name="Person" table="ou=Persons,dc=example,dc=com"
       schema="top,person,organizationalPerson,inetOrgPerson">
    <field name="fullName" primary-key="true" column="cn" />
    <field name="firstName" column="givenName" />
    <field name="lastName" column="sn" />
    <field name="account">
        <embedded null-indicator-column="uid">
            <field name="uid" column="uid" />
            <field name="password" column="userPassword" />
        </embedded>
    </field>
</class>
```

Obviously this strategy brings some limitations. Only one instance of a Java class can be stored embedded, if there would be multiple uid and userPassword attribute it would not be possible to determine which values are associated.

4.4 Queries

The query language of JDO is very powerful. The LDAP store converts the query into a native LDAP search request. Obviously not all query elements could be applied in LDAP, in fact only a subset of the filter expressions are converted to LDAP filters:

- Logical expressions: &, |
- Operators: ==, !=, <, <=, >, >=
- Methods: startsWith(), endsWith()

More advanced filters, aggregation and ordering are handled by the in-memory query evaluator. That could mean that **all** objects of a type are loaded from the directory and evaluated in memory, be aware of that!

4.5 Future Work

The following points should be addressed in future:

- Encryption and strong authentication for LDAP connections.
- Improvements in handling of large data. This may include lazy loading of collections, e.g. using paged search or VLV control.
- Improvements in handling of hierarchical data. Retrieving child objects doesn't work in some cases. DN references are not updated when moving objects within the DIT. JDO Compound Identity may be a good option.
- Improvements in query handling.
- Schema awareness, e.g. to decide if a field could be stored to a particular LDAP attribute type or if a substring or ordering filter could be applied.
- Versioning support to detect concurrent updates. Attributes createTimestamp, modifyTimestamp or entryCSN may be used.
- Support for `java.util.Map`.
- Auto-creation of missing container entries.
- Auto-creation of schema (object classes and attribute type).
- Leverage JDO transactions as soon as there is a standardized transaction mechanism for LDAP.

5 Conclusion

The usage of JDO for LDAP persistence makes sense when the directory server is the datastore for business objects. The developer can concentrate on the business objects and can persist and retrieve these objects using a standardized API. No need to mess around with LDAP. However JDO can't be used in the area where LDAP is strong: doing authentication.

The usage of JDO is also a trade of. It is a mature standard which has been updated multiple times. It also was designed to work in all environments, from embedded systems to large enterprise deployments. Thus the specification is quite complex and the concepts (lifecycles, transactions, configuration parameters) need to be learned. So if only some entries and attributes need to be read from the directory it may not be worth using JDO.

References

[1] JDO Specification, http://db.apache.org/jdo/specifications.html

[2] DataNucleus download site, http://www.datanucleus.org/project/download.html

[3] DataNucleus enhancer, http://www.datanucleus.org/products/accessplatform_2_0/enhancer.html

[4] DataNucleus LDAP store, http://www.datanucleus.org/products/accessplatform_2_0/ldap/support.html