This page last changed on Apr 24, 2006 by ulriksandberg.

# LdapTemplate - Java LDAP Made Simple

## Introduction

There are several similarities between SQL and LDAP programming in Java. Despite being two completely different APIs with different pros and cons, they share a number of less flattering characteristics.

- They require extensive plumbing code even to perform the simplest of tasks.
- All resources need to be correctly closed no matter what happens.
- Exception handling is difficult.

The above points often lead to massive code duplication in common usages of the APIs. As we all know, code duplication is one of the worst code smells. All in all it boils down to this: SQL and LDAP programming in Java are both incredibly dull and repetitive.

Spring JDBC, a part of the Spring framework, provides excellent utilities for simplifying SQL programming. We need a similar framework for LDAP programming.

## Introducing LdapTemplate

LdapTemplate (http://sf.net/projects/ldaptemplate) is a framework for simpler LDAP programming in Java, built on the same principles as Spring JDBC. It completely eliminates the need to worry about creating and closing `LdapContext` and looping through `NamingEnumeration`. It also provides a more comprehensive unchecked Exception hierarchy, built on Spring's DataAccessException. As a bonus, it also contains classes for dynamically building LDAP filters and DNs (Distinguished Names).

## Quick Start and Tutorial

This tutorial will walk you through how to perform different tasks using LdapTemplate. We will start off by going through the basic setup. Then we will show how to perform basic searches, filter and distinguished name management, as well as updates. Towards the end, we will display some powerful additional features that can be extremely useful in some cases.

### The base environment

1. Include the following libraries in your project:
    - commons-logging-1.0.4
    - commons-lang-2.1
    - commons-collections-3.1

- spring-beans-1.2.7
- spring-core-1.2.7
- spring-context-1.2.7
- spring-dao-1.2.7
  (the spring libraries may be replaced with the full spring jar if you already have that in your project.

2. Set up the required beans in your Spring context file and inject the LdapTemplate to your Dao:

```
<beans>
 ...
 <bean id="contextSource" class="net.sf.ldaptemplate.support.LdapContextSource">
   <property name="url" value="ldap://localhost:389" />
   <property name="userName" value="cn=Manager" />
   <property name="base" value="dc=jayway,dc=se" />
   <property name="password" value="secret" />
 </bean>

 <bean id="ldapTemplate" class="net.sf.ldaptemplate.LdapTemplate">
   <constructor-arg ref="contextSource" />
 </bean>
 ...
</beans>
```

3. Add a property and setter for the LdapTemplate in your Dao class and inject the LdapTemplate in the Spring context xml.

```
public class PersonDaoImpl implements PersonDao {
 private LdapTemplate ldapTemplate;

 public void setLdapTemplate(LdapTemplate ldapTemplate) {
   this.ldapTemplate = ldapTemplate;
 }
 ...
}
```

```
<beans>
 ...
 <bean id="personDao" class="se.jayway.dao.PersonDaoImpl">
   <property name="ldapTemplate" ref="ldapTemplate" />
 </bean>
</beans>
```

## Example 1: Searches and Lookups Using AttributesMapper

In this example we will use an `AttributesMapper` to easily build a List of all common names of all person objects.

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  private LdapTemplate ldapTemplate;
  ...
  public List getAllPersonCommonNames() {
    return ldapTemplate.search(
      "", "(objectclass=person)",
      new AttributesMapper() {
          public Object mapFromAttributes(Attributes attrs) throws NamingException {
              return attrs.get("cn").get();
          }
      });
  }
}
```

The inline implementation of `AttributeMapper` just gets the desired attribute value from the `Attributes` and returns it. Internally, `LdapTemplate` iterates over all entries found, calling the given `AttributeMapper` for each entry, and collects the results in a list. The list is then returned by the search.

Note that the `AttributesMapper` implementation could easily be modified to return a full Person object:

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  private class PersonAttributesMapper implements AttributesMapper() {
    public Object mapFromAttributes(Attributes attrs) throws NamingException {
      Person person = new Person();
      person.setFullName((String)attrs.get("cn").get());
      person.setLastName((String)attrs.get("sn").get());
      person.setDescription((String)attrs.get("description").get());
      return person;
    }
  }
}
```

Lookups can be performed in a similar manner:

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  private LdapTemplate ldapTemplate;
  ...
  public Person findPerson(dn) {
    return ldapTemplate.lookup(dn, new PersonAttributesMapper());
  }
}
```

This will lookup the specified DN and pass the found attributes to the supplied `AttributesMapper`, in this case resulting in a `Person` object.

## Example 2: Building Dynamic Distinguished Names (DNs)

Java 1.5 includes an implementation of the `Name` interface, `DistinguishedName`, to simplify working with distinguished names. To help developers that are still in the 1.4-world, we have included our own implementation of `DistinguishedName`. This greatly simplifies building distinguished names, especially what with all the escapings and encodings.

```
package se.jayway.dao;

import net.sf.ldaptemplate.support.DistinguishedName;

public class PersonDaoImpl implements PersonDao {
  private static final String BASE_NAME = "dc=jayway,dc=se";
  ...
  public Name buildDn(Person p) {
    DistinguishedName dn = new DistinguishedName(BASE_NAME);
    dn.append("c", p.getCountry());
    dn.append("ou", p.getCompany());
    dn.append("cn", p.getFullname());
    return dn;
  }
}
```

For a Person with country=Sweden, company=company1 and fullname=Some Person, the result would be the following DN:
`cn=Some Person, ou=company1, c=Sweden, dc=jayway, dc=se`

## Example 3: Binding Data

Inserting data in Java LDAP is called *binding*. In order to do that, a distinguished name that uniquely identifies the new entry is required. The following example shows how data is bound using LdapTemplate:

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  public void create(Person p) {
    Name dn = buildDn(p);
    ldapTemplate.bind(dn, null, buildAttributes(p));
  }

  private Attributes buildAttributes(Person p) {
    Attributes attrs = new BasicAttributes();
    BasicAttribute ocattr = new BasicAttribute("objectclass");
    ocattr.add("top");
    ocattr.add("person");
    attrs.put(ocattr);
    attrs.put("cn", "Some Person");
    attrs.put("sn", "Person");
    return attrs;
  }
}
```

The `Attributes` building is — while dull and verbose — sufficient for many purposes. It is however possible to simplify the binding operation further. This will be described below in Example 6: DirObjectFactory and the DirContextAdapter.

## Example 4: Modifying Data

Modification of data can be done in two ways - using `rebind()` or `modifyAttributes()`:

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  public void update(Person p) {
    Name dn = buildDn(p);
    ldapTemplate.rebind(dn, null, buildAttributes(p));
  }
}
```

or

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  public void update(Person p) {
```

```
      Name dn = buildDn(p);
      Attribute attr = new BasicAttribute("description", p.getDescription());
      ModificationItem item = new ModificationItem(DirContext.REPLACE_ATTRIBUTE, attr);
      ldapTemplate.modifyAttributes(dn, new ModificationItem[] {item});
    }
  }
```

Again, building `Attributes` and `ModificationItem` arrays is a lot of work, but as you will see in Example 6: DirObjectFactory and the DirContextAdapter, the update operations can also be simplified.

## Example 5: Building Dynamic Filters

We can build dynamic filters to use in searches using the classes from the `net.sf.ldaptemplate.support.filter` package.

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  public static final String BASE_DN = "dc=jayway,dc=se";
  ...
  public List getPersonNamesByLastName(String lastName) {
    AndFilter filter = new AndFilter();
    filter.and(new EqualsFilter("objectclass", "person"));
    filter.and(new EqualsFilter("sn", lastName));
    return ldapTemplate.search(
      BASE_DN,
      filter.encode(),
      new AttributesMapper() {
          public Object mapFromAttributes(Attributes attrs) throws NamingException {
              return attrs.get("cn").get();
          }
      });

}
```

To perform a wildcard search, it's possible to use the `WhitespaceWildcardsFilter`:

```
...
    AndFilter filter = new AndFilter();
    filter.and(new EqualsFilter("objectclass", "person"));
    filter.and(new WhitespaceWildcardsFilter("cn", cn));
...
```

If `cn` is `john doe`, the above code would result in the following filter: `(&(objectclass=person)(cn=*john*doe*))`.

## Example 6: DirObjectFactory and the DirContextAdapter

A little-known – and probably underestimated – feature of the Java LDAP API, is the ability to register a `DirObjectFactory` to automatically create objects from found contexts. One of the reasons why it is seldom used is that you will need an implementation of `DirObjectFactory` that creates instances of a meaningful implementation of `DirContext`. The LdapTemplate framework provides the missing pieces: a default implementation of `DirContext` called `DirContextAdapter`, and a corresponding implementation of `DirObjectFactory` called `DefaultDirObjectFactory`. Used together with the `DefaultDirObjectFactory`, the `DirContextAdapter` can be a very powerful tool.

First of all, we need to register the `DefaultDirObjectFactory` with the `ContextSource`. This is done using the dirObjectFactory property:

```
<beans>
  ...
  <bean id="contextSource" class="net.sf.ldaptemplate.support.LdapContextSource" >
    <property name="url" value="ldap://localhost:389" />
    <property name="base" value="dc=jayway,dc=se" />
    <property name="userName" value="cn=Manager" />
    <property name="password" value="secret" />
    <property name="dirObjectFactory"
value="net.sf.ldaptemplate.support.DefaultDirObjectFactory">
  </bean>
</beans>
```

Now, whenever a context is found in the LDAP tree, its `Attributes` will be used to construct a `DirContextAdapter`. This enables us to use a `ContextMapper` to transform found values:

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  private class PersonContextMapper implements ContextMapper {
    public Object mapFromContext(Object ctx) {
      DirContextAdapter context = (DirContextAdapter)ctx;
      Person p = new Person();
      p.setFullName(context.getStringAttribute("cn"));
      p.setLastName(context.getStringAttribute("sn"));
      p.setDescription(context.getStringAttribute("description"));
      return p;
    }
  }

  public Person findPerson(String name, String company, String country) {
    Name dn = buildDn(name, company, country);
    return (Person) ldapTemplate.lookup(dn, new PersonContextMapper());
  }
}
```

The above code shows that it is possible to retrieve the attributes directly by name, without having to go through the `Attributes` and `BasicAttribute` classes.

The `DirContextAdapter` can also be used to hide the `Attributes` when binding and modifying data. This is an example of an improved implementation of the `create` DAO method. Compare it with the previous implementation in Example 3: Binding Data.

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  public void create(Person p) {
    Name dn = buildDn(p);
    DirContextAdapter context= new DirContextAdapter(dn);

    context.setAttributeValues("objectclass", new String[]{"top", "person"});
    context.setAttributeValue("cn", p.getFullname());
    context.setAttributeValue("sn", p.getLastname());
    context.setAttributeValue("description",
         p.getDescription());
    ldapTemplate.bind(dn, adapter, null);
  }
}
```

Obviously, the code would be pretty much identical for a rebind. However, let's say that you don't want to remove and re-create the entry, but instead update only the attributes that have changed. The `DirContextAdapter` also has the ability to keep track of its modified attributes. The following example takes advantage of this feature:

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  public void update(Person p) {
    Name dn = buildDn(p);
    DirContextAdapter context = (DirContextAdapter)ldapTemplate.lookup(dn);

    context.setAttributeValues("objectclass", new String[] {"top", "person"});
    context.setAttributeValue("cn", p.getFullname());
    context.setAttributeValue("sn", p.getLastname());
    context.setAttributeValue("description", p.getDescription());

    ldapTemplate.modifyAttributes(dn, context.getModificationItems());
  }
}
```

## Example 7: Missing Overloaded API Methods

While `LdapTemplate` contains several overloaded versions of the most common operations, we have not provided an alternative for each and every method signature in LdapContext, mostly because there are so many of them. We have, however, provided a means to call whichever overloaded method you want. Let's say, for example, that you want to use the method `search(Name name, String filterExpr, Object[] filterArgs, SearchControls ctls)`. The way to do this is to use a custom `SearchExecutor` implementation:

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  public List search(final Name base, final String filter,
      final String[] params, final SearchControls ctls) {
    SearchExecutor se = new SearchExecutor() {
      public NamingEnumeration executeSearch(LdapContext ctx) {
        return ctx.search(base, filter, params, ctls);
      }
    };

    SearchResultCallbackHandler handler =
      ldapTemplate.new AttributesMapperCallbackHandler(new PersonAttributesMapper());

    return ldapTemplate.search(se, handler);
  }
}
```

or

```
package se.jayway.dao;

public class PersonDaoImpl implements PersonDao {
  ...
  public List search(final Name base, final String filter,
      final String[] params, final SearchControls ctls){
    SearchExecutor se = new SearchExecutor() {
      public NamingEnumeration executeSearch(LdapContext ctx) {
        return ctx.search(base, filter, params, ctls);
      }
```

```
      };

      SearchResultCallbackHandler handler =
        ldapTemplate.new ContextMapperCallbackHandler(new PersonContextMapper());

      return ldapTemplate.search(se, handler);
    }
  }
```

Note that when using the `LdapTemplate.ContextMapperCallbackHandler` you **must** make sure that you have called `setReturningObjFlag(true)` on your `SearchControls` instance.

In the same manner, you can execute any method on a context by using a `ContextExecutor` and the `executeReadOnly()` and `executeReadWrite()` methods.

## Conclusion and Further Directions

By now it should be clear that the LdapTemplate framework will be of great help for any Java project that communicates with LDAP. For further examples, consult the integration unit tests in the source (`src/iutest`) and the Javadocs. If you have any comments, problems or questions, please let us know. Subscribe to our mailing list <http://lists.sourceforge.net/lists/listinfo/ldaptemplate-user>, or just drop us a note <ldaptemplate-user@lists.sourceforge.net>.