Spring LDAP - Reference Documentation

MattiasArthursson, UlrikSandberg, EricDalquist, KeithBarlow

Table of Contents

Preface

The Java Naming and Directory Interface (JNDI) is for LDAP programming what Java Database Connectivity (JDBC) is for SQL programming. There are several similarities between JDBC and JNDI/LDAP (Java LDAP). 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: JDBC 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 Java LDAP programming.

1. Introduction

1.1. Overview

Spring LDAP (http://www.springframework.org/ldap) is a library for simpler LDAP programming in Java, built on the same principles as the JdbcTemplate in 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), LDAP attribute management, and client-side LDAP transaction management.

Consider, for example, a method that should search some storage for all persons and return their names in a list. Using JDBC, we would create a *connection* and execute a *query* using a *statement*. We would then loop over the *result set* and retrieve the *column* we want, adding it to a list. In contrast, using Java LDAP, we would create a *context* and perform a *search* using a *search filter*. We would then loop over the resulting *naming enumeration* and retrieve the *attribute* we want, adding it to a list.

The traditional way of implementing this person name search method in Java LDAP looks like this, where the code marked as bold actually performs tasks related to the business purpose of the method:

```
package com.example.dao;

public class TraditionalPersonDaoImpl implements PersonDao {
   public List getAllPersonNames() {
      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
      env.put(Context.PROVIDER_URL, "ldap://localhost:389/dc=example,dc=com");

      DirContext ctx;
      try {
         ctx = new InitialDirContext(env);
      } catch (NamingException e) {
         throw new RuntimeException(e);
      }

      LinkedList list = new LinkedList();
      NamingEnumeration results = null;
      try {
         SearchControls controls = new SearchControls();
         controls.setSearchScope(SearchControls.SUBTREE_SCOPE);
         results = ctx.search("", "(objectclass=person)", controls);

         while (results.hasMore()) {
            SearchResult searchResult = (SearchResult) results.next();
            Attributes attributes = searchResult.getAttributes();
            Attribute attr = attributes.get("cn");
            String cn = (String) attr.get();
            list.add(cn);
         }
      } catch (NameNotFoundException e) {
         // The base context was not found.
         // Just clean up and exit.
      } catch (NamingException e) {
         throw new RuntimeException(e);
      } finally {
         if (results != null) {
            try {
               results.close();
            } catch (Exception e) {
               // Never mind this.
            }
         }
         if (ctx != null) {
            try {
               ctx.close();
            } catch (Exception e) {
               // Never mind this.
            }
         }
      }
      return list;
   }
}
```

By using the Spring LDAP classes `AttributesMapper` and `LdapTemplate`, we get the exact same functionality with the following code:

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   private LdapTemplate ldapTemplate;

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

   public List getAllPersonNames() {
      return ldapTemplate.search(
         "", "(objectclass=person)",
         new AttributesMapper() {
            public Object mapFromAttributes(Attributes attrs)
               throws NamingException {
               return attrs.get("cn").get();
            }
         });
   }
}
```

The amount of boiler-plate code is significantly less than in the traditional example. The `LdapTemplate` version of the search method performs the search, maps the attributes to a string using the given `AttributesMapper`, collects the strings in an internal list, and finally returns the list.

Note that the `PersonDaoImpl` code simply assumes that it has an `LdapTemplate` instance, rather than looking one up somewhere. It provides a set method for this purpose. There is nothing Spring-specific about this "Inversion of Control". Anyone that can create an instance of `PersonDaoImpl` can also set the `LdapTemplate` on it. However, Spring provides a very flexible and easy way of achieving this. The Spring container can be told to wire up an instance of `LdapTemplate` with its required dependencies and inject it into the `PersonDao` instance. This wiring can be defined in various ways, but the most common is through XML:

```
<beans>
   <bean id="contextSource"
 class="org.springframework.ldap.core.support.LdapContextSource">
      <property name="url" value="ldap://localhost:389" />
      <property name="base" value="dc=example,dc=com" />
      <property name="userDn" value="cn=Manager" />
      <property name="password" value="secret" />
   </bean>

   <bean id="ldapTemplate" class="org.springframework.ldap.core.LdapTemplate">
      <constructor-arg ref="contextSource" />
   </bean>

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

1.2. Packaging overview

At a minimum, to use Spring LDAP you need:

- *spring-ldap-core* (the Spring LDAP library)
- *spring-core* (miscellaneous utility classes used internally by the framework)
- *spring-beans* (contains interfaces and classes for manipulating Java beans)

- *commons-logging* (a simple logging facade, used internally)
- *commons-lang* (misc utilities, used internally)

In addition to the required dependencies the following optional dependencies are required for certain functionality:

- *spring-context* (If your application is wired up using the Spring Application Context - adds the ability for application objects to obtain resources using a consistent API. Definitely needed if you are planning on using the BaseLdapPathBeanPostProcessor.)
- *spring-tx* (If you are planning to use the client side compensating transaction support)
- *spring-jdbc* (If you are planning to use the client side compensating transaction support)
- *ldapbp* (Sun LDAP Booster Pack - if you will use the LDAP v3 Server controls integration and you're not using Java5 or higher)
- *commons-pool* (If you are planning to use the pooling functionality)
- *spring-batch* (If you are planning to use the LDIF parsing functionality together with Spring Batch)

1.3. Package structure

This section provides an overview of the logical package structure of the Spring LDAP codebase. The dependencies for each package are clearly noted.

Figure 1.1. Spring LDAP package structure

org.springframework.transaction.compensating

The *transaction.compensating* package contains the generic compensating transaction support. This is not LDAP-specific or JNDI-specific in any way.

- Dependencies: commons-logging

org.springframework.ldap

The *ldap* package contains the exceptions of the library. These exceptions form an unchecked hierarchy that mirrors the NamingException hierarchy.

- Dependencies: spring-core

org.springframework.ldap.core

The *ldap.core* package contains the central abstractions of the library. These abstractions include AuthenticationSource, ContextSource, DirContextProcessor, and NameClassPairCallbackHandler. This package also contains the central class LdapTemplate, plus various mappers and executors.

- Dependencies: ldap, ldap.support, spring-beans, spring-core, spring-tx, commons-lang, commons-logging

org.springframework.ldap.core.support

The *ldap.core.support* package contains supporting implementations of some of the core interfaces.

- Dependencies: ldap, ldap.core, ldap.support, spring-core, spring-beans, spring-context, commons-lang, commons-logging

org.springframework.ldap.core.simple

The *ldap.core.simple* package contains Java5-specific parts of Spring LDAP. It's mainly a simplification layer that takes advantage of the generics support in Java5, in order to get typesafe context mappers as well as typesafe search and lookup methods.

- Dependencies: ldap.core

org.springframework.ldap.pool

The *ldap.pool* package contains support for detailed pool configuration on a per-ContextSource basis. Pooling support is provided by PoolingContextSource which can wrap any ContextSource and pool both read-only and read-write DirContext objects. Jakarta Commons-Pool is used to provide the underlying pool implementation.

- Dependencies: ldap.core, commons-lang, commons-pool

org.springframework.ldap.pool.factory

The *ldap.pool.factory* package contains the actual pooling context source and other classes for context creation.

- Dependencies: ldap, ldap.core, ldap.pool, ldap.pool.validation, spring-beans, spring-tx, commons-lang, commons-logging, commons-pool

org.springframework.ldap.pool.validation

The *ldap.pool.validation* package contains the connection validation support.

- Dependencies: ldap.pool, commons-lang, commons-logging

org.springframework.ldap.support

The *ldap.support* package contains supporting utilities, like the exception translation mechanism.

- Dependencies: ldap, spring-core, commons-lang, commons-logging

org.springframework.ldap.authentication

The *ldap.authentication* package contains an implementation of the AuthenticationSource interface that can be used if the user should be allowed to read some information even though not logged in.

- Dependencies: ldap.core, spring-beans, commons-lang

org.springframework.ldap.control

The *ldap.control* package contains an abstract implementation of the DirContextProcessor interface that can be used as a basis for processing RequestControls and ResponseControls. There is also a concrete implementation that handles paged search results and one that handles sorting. The LDAP Booster Pack is used to get support for controls, unless Java5 is used.

- Dependencies: ldap, ldap.core, LDAP booster pack (optional), spring-core, commons-lang, commons-logging

org.springframework.ldap.filter

The *ldap.filter* package contains the Filter abstraction and several implementations of it.

- Dependencies: ldap.core, spring-core, commons-lang

org.springframework.ldap.transaction.compensating

The *ldap.transaction.compensating* package contains the core LDAP-specific implementation of compensating transactions.

- Dependencies: ldap.core, ldap.core.support, transaction.compensating, spring-core, commons-lang, commons-logging

org.springframework.ldap.transaction.compensating.manager

The *ldap.transaction.compensating.manager* package contains the core implementation classes for client-side compensating transactions.

- Dependencies: ldap, ldap.core, ldap.support, ldap.transaction.compensating, ldap.transaction.compensating.support, transaction.compensating, spring-tx, spring-jdbc, spring-orm, commons-logging

org.springframework.ldap.transaction.compensating.support

The *ldap.transaction.compensating.support* package contains useful helper classes for client-side compensating transactions.

- Dependencies: ldap.core, ldap.transaction.compensating

org.springframework.ldap.ldif

The ldap.ldif package provides support for parsing LDIF files.

- Dependencies: ldap.core

org.springframework.ldap.ldif.batch

The ldap.ldif.batch package provides the classes necessary to use the LDIF parser in the Spring Batch framework.

- Dependencies: ldap.core, ldap.ldif.parser, spring-batch, spring-core, spring-beans, commons-logging

org.springframework.ldap.ldif.parser

The ldap.ldif.parser package provides the parser classes and interfaces.

- Dependencies: ldap.core, ldap.schema, ldap.ldif, ldap.ldif.support, spring-core, spring-beans, commons-lang, commons-logging

org.springframework.ldap.ldif.support

The ldap.ldif.support package provides the necessary auxiliary classes utilized by the LDIF Parser.

- Dependencies: ldap.core, ldap.ldif, commons-lang, commons-logging

org.springframework.ldap.odm

The ldap.odm package provides the classes and interfaces enabling annotation based object-directory mapping.

- Dependencies: ldap, ldap.core, ldap.core.simple, ldap.filter, spring-beans, commons-cli, commons-logging, freemarker

For the exact list of jar dependencies, see the Spring LDAP Maven2 Project Object Model (POM) files in the source tree.

1.4. Support

Spring LDAP 1.3 is supported on Spring 2.0 and later.

The community support forum is located at http://forum.springframework.org, and the project web page is http://www.springframework.org/ldap.

2. Basic Operations

2.1. Search and Lookup Using AttributesMapper

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

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   private LdapTemplate ldapTemplate;

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

   public List getAllPersonNames() {
      return ldapTemplate.search(
         "", "(objectclass=person)",
         new AttributesMapper() {
            public Object mapFromAttributes(Attributes attrs)
               throws NamingException {
               return attrs.get("cn").get();
            }
         });
   }
}
```

Example 2.1 AttributesMapper that returns a single attribute

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

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

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   private LdapTemplate ldapTemplate;
   ...
   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;
      }
   }

   public List getAllPersons() {
      return ldapTemplate.search("", "(objectclass=person)", new
 PersonAttributesMapper());
   }
}
```

Example 2.2 AttributesMapper that returns a Person object

If you have the distinguished name (dn) that identifies an entry, you can retrieve the entry directly, without searching for it. This is called a *lookup* in Java LDAP. The following example shows how a lookup results in a Person object:

```
package com.example.dao;

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

Example 2.3 A lookup resulting in a Person object

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

2.2. Building Dynamic Filters

We can build dynamic filters to use in searches, using the classes from the `org.springframework.ldap.filter` package. Let's say that we want the following filter: `(&(objectclass=person)(sn=?))`, where we want the `?` to be replaced with the value of the parameter `lastName`. This is how we do it using the filter support classes:

```
package com.example.dao;

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

Example 2.4 Building a search filter dynamically

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));
```

Example 2.5 Building a wildcard search filter

Note

In addition to simplifying building of complex search filters, the `Filter` classes also provide proper escaping of any unsafe characters. This prevents "ldap injection", where a user might use such characters to inject unwanted operations into your LDAP operations.

2.3. Building Dynamic Distinguished Names

The standard Name interface represents a generic name, which is basically an ordered sequence of components. The `Name` interface also provides operations on that sequence; e.g., `add` or `remove`. LdapTemplate provides an implementation of the `Name` interface: `DistinguishedName`. Using this class will greatly simplify building distinguished names, especially considering the sometimes complex rules regarding escapings and encodings. As with the `Filter` classes this helps preventing potentially malicious data being injected into your LDAP operations.

The following example illustrates how `DistinguishedName` can be used to dynamically construct a distinguished name:

```
package com.example.dao;

import org.springframework.ldap.core.support.DistinguishedName;
import javax.naming.Name;

public class PersonDaoImpl implements PersonDao {
   public static final String BASE_DN = "dc=example,dc=com";
   ...
   protected Name buildDn(Person p) {
      DistinguishedName dn = new DistinguishedName(BASE_DN);
      dn.add("c", p.getCountry());
      dn.add("ou", p.getCompany());
      dn.add("cn", p.getFullname());
      return dn;
   }
}
```

Example 2.6 Building a distinguished name dynamically

Assuming that a Person has the following attributes:

`country`	Sweden
`company`	Some Company
`fullname`	Some Person

The code above would then result in the following distinguished name:

```
cn=Some Person, ou=Some Company, c=Sweden, dc=example, dc=com
```

In Java 5, there is an implementation of the Name interface: LdapName. If you are in the Java 5 world, you might as well use `LdapName`. However, you may still use `DistinguishedName` if you so wish.

2.4. Binding and Unbinding

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 com.example.dao;

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

Example 2.7 Binding data using Attributes

The Attributes building is--while dull and verbose--sufficient for many purposes. It is, however, possible to simplify the binding operation further, which will be described in Chapter 3, *Simpler Attribute Access and Manipulation with DirContextAdapter*.

Unbinding Data

Removing data in Java LDAP is called unbinding. A distinguished name (dn) is required to identify the entry, just as in the binding operation. The following example shows how data is unbound using LdapTemplate:

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   private LdapTemplate ldapTemplate;
   ...
   public void delete(Person p) {
      Name dn = buildDn(p);
      ldapTemplate.unbind(dn);
   }
}
```

Example 2.8 Unbinding data

2.5. Modifying

In Java LDAP, data can be modified in two ways: either using *rebind* or *modifyAttributes*.

Modifying using `rebind`

A `rebind` is a very crude way to modify data. It's basically an `unbind` followed by a `bind`. It looks like this:

```
package com.example.dao;

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

Example 2.9 Modifying using rebind

Modifying using `modifyAttributes`

If only the modified attributes should be replaced, there is a method called `modifyAttributes` that takes an array of modifications:

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   private LdapTemplate ldapTemplate;
   ...
   public void updateDescription(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});
   }
}
```

Example 2.10 Modifying using modifyAttributes

Building `Attributes` and `ModificationItem` arrays is a lot of work, but as you will see in Chapter 3, *Simpler Attribute Access and Manipulation with DirContextAdapter*, the update operations can be simplified.

2.6. Sample applications

It is recommended that you review the Spring LDAP sample applications included in the release distribution for best-practice illustrations of the features of this library. A description of each sample is provided below:

1. spring-ldap-person - the sample demonstrating most features.

2. spring-ldap-article - the sample application that was written to accompany a java.net article about Spring LDAP.

3. Simpler Attribute Access and Manipulation with DirContextAdapter

3.1. Introduction

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 Spring LDAP library provides the missing pieces: a default implementation of `DirContext` called `DirContextAdapter`, and a corresponding implementation of `DirObjectFactory` called `DefaultDirObjectFactory`. Used together with `DefaultDirObjectFactory`, the `DirContextAdapter` can be a very powerful tool.

3.2. Search and Lookup Using ContextMapper

The `DefaultDirObjectFactory` is registered with the `ContextSource` by default, which means that whenever a context is found in the LDAP tree, its `Attributes` and Distinguished Name (DN) will be used to construct a `DirContextAdapter`. This enables us to use a `ContextMapper` instead of an `AttributesMapper` to transform found values:

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   ...
   private static 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 findByPrimaryKey(
      String name, String company, String country) {
      Name dn = buildDn(name, company, country);
      return ldapTemplate.lookup(dn, new PersonContextMapper());
   }
}
```

Example 3.1 Searching using a ContextMapper

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. This is particularly useful when working with multi-value attributes. Extracting values from multi-value attributes normally requires looping through a `NamingEnumeration` of attribute values returned from the `Attributes` implementation. The `DirContextAdapter` can do this for you, using the `getStringAttributes()` or `getObjectAttributes()` methods:

```
private static 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"));
      // The roleNames property of Person is an String array
      p.setRoleNames(context.getStringAttributes("roleNames"));
      return p;
   }
}
```

Example 3.2 Getting multi-value attribute values using `getStringAttributes()`

The AbstractContextMapper

Spring LDAP provides an abstract base implementation of `ContextMapper`, `AbstractContextMapper`. This automatically takes care of the casting of the supplied `Object` parameter to `DirContexOperations`. The `PersonContextMapper` above can thus be re-written as follows:

```

  private static class PersonContextMapper extends AbstractContextMapper {
      public Object doMapFromContext(DirContextOperations ctx) {
         Person p = new Person();
         p.setFullName(context.getStringAttribute("cn"));
         p.setLastName(context.getStringAttribute("sn"));
         p.setDescription(context.getStringAttribute("description"));
         return p;
      }
  }
```

Example 3.3 Using an AbstractContextMapper

3.3. Binding and Modifying Using DirContextAdapter

While very useful when extracting attribute values, `DirContextAdapter` is even more powerful for hiding attribute details when binding and modifying data.

Binding

This is an example of an improved implementation of the create DAO method. Compare it with the previous implementation in the section called “Binding Data”.

```
package com.example.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(context);
   }
}
```

Example 3.4 Binding using `DirContextAdapter`

Note that we use the `DirContextAdapter` instance as the second parameter to bind, which should be a `Context`. The third parameter is `null`, since we're not using any `Attributes`.

Also note the use of the `setAttributeValues()` method when setting the `objectclass` attribute values. The `objectclass` attribute is multi-value, and similar to the troubles of extracting muti-value attribute data, building multi-value attributes is tedious and verbose work. Using the `setAttributeValues()` mehtod you can have `DirContextAdapter` handle that work for you.

Modifying

The code for a `rebind` would be pretty much identical to Example 3.4, “Binding using `DirContextAdapter`”, except that the method called would be `rebind`. As we saw in the section called “Modifying using `modifyAttributes`” a more correct approach would be to build a `ModificationItem` array containing the actual modifications you want to do. This would require you to determine the actual modifications compared to the data present in the LDAP tree. Again, this is something that `DirContextAdapter` can help you with; the `DirContextAdapter` has the ability to keep track of its modified attributes. The following example takes advantage of this feature:

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   ...
   public void update(Person p) {
      Name dn = buildDn(p);
      DirContextOperations context = ldapTemplate.lookupContext(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(context);
   }
}
```

Example 3.5 Modifying using `DirContextAdapter`

When no mapper is passed to a `ldapTemplate.lookup()` operation, the result will be a `DirContextAdapter` instance. While the `lookup` method returns an `Object`, the convenience method `lookupContext` method automatically casts the return value to a `DirContextOperations` (the interface that `DirContextAdapter` implements.

The observant reader will see that we have duplicated code in the `create` and `update` methods. This code maps from a domain object to a context. It can be extracted to a separate method:

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   private LdapTemplate ldapTemplate;

   ...
   public void create(Person p) {
      Name dn = buildDn(p);
      DirContextAdapter context = new DirContextAdapter(dn);
      mapToContext(p, context);
      ldapTemplate.bind(context);
   }

   public void update(Person p) {
      Name dn = buildDn(p);
      DirContextOperations context = ldapTemplate.lookupContext(dn);
      mapToContext(person, context);
      ldapTemplate.modifyAttributes(context);
   }

   protected void mapToContext (Person p, DirContextOperations context) {
      context.setAttributeValues("objectclass", new String[] {"top", "person"});
      context.setAttributeValue("cn", p.getFullName());
      context.setAttributeValue("sn", p.getLastName());
      context.setAttributeValue("description", p.getDescription());
   }
}
```

Example 3.6 Binding and modifying using DirContextAdapter

3.4. A Complete PersonDao Class

To illustrate the power of Spring LDAP, here is a complete Person DAO implementation for LDAP in just 68 lines:

```
package com.example.dao;

import java.util.List;

import javax.naming.Name;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;

import org.springframework.ldap.core.AttributesMapper;
import org.springframework.ldap.core.ContextMapper;
import org.springframework.ldap.core.LdapTemplate;
import org.springframework.ldap.core.DirContextAdapter;
import org.springframework.ldap.core.support.DistinguishedName;
import org.springframework.ldap.filter.AndFilter;
import org.springframework.ldap.filter.EqualsFilter;
import org.springframework.ldap.filter.WhitespaceWildcardsFilter;

public class PersonDaoImpl implements PersonDao {
   private LdapTemplate ldapTemplate;

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

   public void create(Person person) {
      DirContextAdapter context = new DirContextAdapter(buildDn(person));
      mapToContext(person, context);
      ldapTemplate.bind(context);
   }

   public void update(Person person) {
      Name dn = buildDn(person);
      DirContextOperations context = ldapTemplate.lookupContext(dn);
      mapToContext(person, context);
      ldapTemplate.modifyAttributes(context);
   }

   public void delete(Person person) {
      ldapTemplate.unbind(buildDn(person));
   }

   public Person findByPrimaryKey(String name, String company, String country) {
      Name dn = buildDn(name, company, country);
      return (Person) ldapTemplate.lookup(dn, getContextMapper());
   }

   public List findByName(String name) {
      AndFilter filter = new AndFilter();
      filter.and(new EqualsFilter("objectclass", "person")).and(new
 WhitespaceWildcardsFilter("cn",name));
      return ldapTemplate.search(DistinguishedName.EMPTY_PATH, filter.encode(),
 getContextMapper());
   }

   public List findAll() {
      EqualsFilter filter = new EqualsFilter("objectclass", "person");
      return ldapTemplate.search(DistinguishedName.EMPTY_PATH, filter.encode(),
 getContextMapper());
   }

   protected ContextMapper getContextMapper() {
      return new PersonContextMapper();
   }

   protected Name buildDn(Person person) {
      return buildDn(person.getFullname(), person.getCompany(), person.getCountry());
   }

   protected Name buildDn(String fullname, String company, String country) {
```

Note

In several cases the Distinguished Name (DN) of an object is constructed using properties of the object. E.g. in the above example, the country, company and full name of the `Person` are used in the DN, which means that updating any of these properties will actually require moving the entry in the LDAP tree using the `rename()` operation in addition to updating the `Attribute` values. Since this is highly implementation specific this is something you'll need to keep track of yourself - either by disallowing the user to change these properties or performing the `rename()` operation in your `update()` method if needed.

4. Adding Missing Overloaded API Methods

4.1. Implementing Custom Search Methods

While `LdapTemplate` contains several overloaded versions of the most common operations in `DirContext`, we have not provided an alternative for each and every method signature, mostly because there are so many of them. We have, however, provided a means to call whichever `DirContext` method you want and still get the benefits that LdapTemplate provides.

Let's say that you want to call the following `DirContext` method:

```
NamingEnumeration search(Name name, String filterExpr, Object[] filterArgs, SearchControls
 ctls)
```

There is no corresponding overloaded method in LdapTemplate. The way to solve this is to use a custom `SearchExecutor` implementation:

```
public interface SearchExecutor {
   public NamingEnumeration executeSearch(DirContext ctx) throws NamingException;
}
```

In your custom executor, you have access to a `DirContext` object, which you use to call the method you want. You then provide a handler that is responsible for mapping attributes and collecting the results. You can for example use one of the available implementations of `CollectingNameClassPairCallbackHandler`, which will collect the mapped results in an internal list. In order to actually execute the search, you call the `search` method in LdapTemplate that takes an executor and a handler as arguments. Finally, you return whatever your handler has collected.

```
package com.example.dao;

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

      CollectingNameClassPairCallbackHandler handler =
         new AttributesMapperCallbackHandler(new PersonAttributesMapper());

      ldapTemplate.search(executor, handler);
      return handler.getList();
   }
}
```

Example 4.1 A custom search method using SearchExecutor and AttributesMapper

If you prefer the `ContextMapper` to the `AttributesMapper`, this is what it would look like:

```
package com.example.dao;

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

      CollectingNameClassPairCallbackHandler handler =
         new ContextMapperCallbackHandler(new PersonContextMapper());

      ldapTemplate.search(executor, handler);
      return handler.getList();
   }
}
```

Example 4.2 A custom search method using SearchExecutor and ContextMapper

Note

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

4.2. Implementing Other Custom Context Methods

In the same manner as for custom `search` methods, you can actually execute any method in `DirContext` by using a `ContextExecutor`.

```
public interface ContextExecutor {
   public Object executeWithContext(DirContext ctx) throws NamingException;
}
```

When implementing a custom `ContextExecutor`, you can choose between using the `executeReadOnly()` or the `executeReadWrite()` method. Let's say that we want to call this method:

```
Object lookupLink(Name name)
```

It's available in `DirContext`, but there is no matching method in `LdapTemplate`. It's a lookup method, so it should be read-only. We can implement it like this:

```
package com.example.dao;

public class PersonDaoImpl implements PersonDao {
   ...
   public Object lookupLink(final Name name) {
      ContextExecutor executor = new ContextExecutor() {
         public Object executeWithContext(DirContext ctx) {
            return ctx.lookupLink(name);
         }
      };

      return ldapTemplate.executeReadOnly(executor);
   }
}
```

In the same manner you can execute a read-write operation using the `executeReadWrite()` method.

Example 4.3 A custom DirContext method using ContextExecutor

5. Processing the DirContext

5.1. Custom DirContext Pre/Postprocessing

In some situations, one would like to perform operations on the `DirContext` before and after the search operation. The interface that is used for this is called `DirContextProcessor`:

```
public interface DirContextProcessor {
   public void preProcess(DirContext ctx) throws NamingException;
   public void postProcess(DirContext ctx) throws NamingException;
}
```

The `LdapTemplate` class has a search method that takes a `DirContextProcessor`:

```
public void search(SearchExecutor se, NameClassPairCallbackHandler handler,
   DirContextProcessor processor) throws DataAccessException;
```

Before the search operation, the `preProcess` method is called on the given `DirContextProcessor` instance. After the search has been executed and the resulting `NamingEnumeration` has been processed, the `postProcess` method is called. This enables a user to perform operations on the `DirContext` to be used in the search, and to check the `DirContext` when the search has been performed. This can be very useful for example when handling request and response controls.

There are also a few convenience methods for those that don't need a custom `SearchExecutor`:

```
public void search(Name base, String filter,
   SearchControls controls, NameClassPairCallbackHandler handler, DirContextProcessor
 processor)

public void search(String base, String filter,
   SearchControls controls, NameClassPairCallbackHandler handler, DirContextProcessor
 processor)

public void search(Name base, String filter,
   SearchControls controls, AttributesMapper mapper, DirContextProcessor processor)

public void search(String base, String filter,
   SearchControls controls, AttributesMapper mapper, DirContextProcessor processor)

public void search(Name base, String filter,
   SearchControls controls, ContextMapper mapper, DirContextProcessor processor)

public void search(String base, String filter,
   SearchControls controls, ContextMapper mapper, DirContextProcessor processor)
```

5.2. Implementing a Request Control DirContextProcessor

The LDAPv3 protocol uses Controls to send and receive additional data to affect the behavior of predefined operations. In order to simplify the implementation of a request control `DirContextProcessor`, Spring LDAP provides the base class `AbstractRequestControlDirContextProcessor`. This class handles the retrieval of the current request controls from the `LdapContext`, calls a template method for creating a request control, and adds it to the `LdapContext`. All you have to do in the subclass is to implement the template method `createRequestControl`, and of course the `postProcess` method for performing whatever you need to do after the search.

```
public abstract class AbstractRequestControlDirContextProcessor implements
      DirContextProcessor {

   public void preProcess(DirContext ctx) throws NamingException {
      ...
   }

   public abstract Control createRequestControl();
}
```

A typical `DirContextProcessor` will be similar to the following:

```
package com.example.control;

public class MyCoolRequestControl extends AbstractRequestControlDirContextProcessor {
   private static final boolean CRITICAL_CONTROL = true;
   private MyCoolCookie cookie;
   ...
   public MyCoolCookie getCookie() {
      return cookie;
   }

   public Control createRequestControl() {
      return new SomeCoolControl(cookie.getCookie(), CRITICAL_CONTROL);
   }

   public void postProcess(DirContext ctx) throws NamingException {
      LdapContext ldapContext = (LdapContext) ctx;
      Control[] responseControls = ldapContext.getResponseControls();

      for (int i = 0; i < responseControls.length; i++) {
         if (responseControls[i] instanceof SomeCoolResponseControl) {
            SomeCoolResponseControl control = (SomeCoolResponseControl)
 responseControls[i];
            this.cookie = new MyCoolCookie(control.getCookie());
         }
      }
   }
}
```

Example 5.1 A request control DirContextProcessor implementation

Note

Make sure you use `LdapContextSource` when you use Controls. The `Control` interface is specific for LDAPv3 and requires that `LdapContext` is used instead of `DirContext`. If an `AbstractRequestControlDirContextProcessor` subclass is called with an argument that is not an `LdapContext`, it will throw an `IllegalArgumentException`.

5.3. Paged Search Results

Some searches may return large numbers of results. When there is no easy way to filter out a smaller amount, it would be convenient to have the server return only a certain number of results each time it is called. This is known as *paged search results*. Each "page" of the result could then be displayed at the time, with links to the next and previous page. Without this functionality, the client must either manually limit the search result into pages, or retrieve the whole result and then chop it into pages of suitable size. The former would be rather complicated, and the latter would be consuming unnecessary amounts of memory.

Some LDAP servers have support for the `PagedResultsControl`, which requests that the results of a search operation are returned by the LDAP server in pages of a specified size. The user controls the rate at which the pages are returned, simply by the rate at which the searches are called. However, the user must keep track of a *cookie* between the calls. The server uses this cookie to keep track of where it left off the previous time it was called with a paged results request.

Spring LDAP provides support for paged results by leveraging the concept for pre- and postprocessing of an `LdapContext` that was discussed in the previous sections. It does so by providing two classes: `PagedResultsRequestControl` and `PagedResultsCookie`. The `PagedResultsRequestControl` class creates a `PagedResultsControl` with the requested page size and adds it to the `LdapContext`. After the search, it gets the `PagedResultsResponseControl` and retrieves two pieces of information from it: the estimated total result size and a cookie. This cookie is a byte array containing information that the server needs the next time it is called with a `PagedResultsControl`. In order to make it easy to store this cookie between searches, Spring LDAP provides the wrapper class `PagedResultsCookie`.

Below is an example of how the paged search results functionality may be used:

```
public PagedResult getAllPersons(PagedResultsCookie cookie) {
   PagedResultsRequestControl control = new PagedResultsRequestControl(PAGE_SIZE, cookie);
   SearchControls searchControls = new SearchControls();
   searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);

   List persons = ldapTemplate.search("", "objectclass=person", searchControls, control);

   return new PagedResult(persons, control.getCookie());
 }
```

Example 5.2 Paged results using `PagedResultsRequestControl`

In the first call to this method, `null` will be supplied as the cookie parameter. On subsequent calls the client will need to supply the cookie from the last search (returned wrapped in the `PagedResult`) each time the method is called. When the actual cookie is `null` (i.e. `pagedResult.getCookie().getCookie()` returns `null`), the last batch has been returned from the search.

6. Transaction Support

6.1. Introduction

Programmers used to working with relational databases coming to the LDAP world often express surprise to the fact that there is no notion of transactions. It is not specified in the protocol, and thus no servers support it. Recognizing that this may be a major problem, Spring LDAP provides support for client-side, compensating transactions on LDAP resources.

LDAP transaction support is provided by `ContextSourceTransactionManager`, a `PlatformTransactionManager` implementation that manages Spring transaction support for LDAP operations. Along with its collaborators it keeps track of the LDAP operations performed in a transaction, making record of the state before each operation and taking steps to restore the initial state should the transaction need to be rolled back.

In addition to the actual transaction management, Spring LDAP transaction support also makes sure that the same `DirContext` instance will be used throughout the same transaction, i.e. the `DirContext` will not actually be closed until the transaction is finished, allowing for more efficient resources usage.

Note

It is important to note that while the approach used by Spring LDAP to provide transaction support is sufficient for many cases it is by no means "real" transactions in the traditional sense. The server is completely unaware of the transactions, so e.g. if the connection is broken there will be no hope to rollback the transaction. While this should be carefully considered it should also be noted that the alternative will be to operate without any transaction support whatsoever; this is pretty much as good as it gets.

Note

The client side transaction support will add some overhead in addition to the work required by the original operations. While this overhead should not be something to worry about in most cases, if your application will not perform several LDAP operations within the same transaction (e.g. a `modifyAttributes` followed by a `rebind`), or if transaction synchronization with a JDBC data source is not required (see below) there will be nothing to gain by using the LDAP transaction support.

Note

While the default setup will work fine for most simple use cases, some more complex scenarios will require additional configuration; more specifically if you will be creating or deleting subtrees within transactions, you will need to use an alternative `TempEntryRenamingStrategy`, as described in the section called “Renaming Strategies” below

6.2. Configuration

Configuring Spring LDAP transactions should look very familiar if you're used to configuring Spring transactions. You will create a `TransactionManager` instance and wrap your target object using a `TransactionProxyFactoryBean`. In addition to this, you will also need to wrap your `ContextSource` in a `TransactionAwareContextSourceProxy`.

```
<beans>
   ...
   <bean id="contextSourceTarget"
 class="org.springframework.ldap.core.support.LdapContextSource">
      <property name="url" value="ldap://localhost:389" />
      <property name="base" value="dc=example,dc=com" />
      <property name="userDn" value="cn=Manager" />
      <property name="password" value="secret" />
   </bean>

   <bean id="contextSource"

 class="org.springframework.ldap.transaction.compensating.manager.TransactionAwareContextSourceProxy">
      <constructor-arg ref="contextSourceTarget" />
   </bean>

   <bean id="ldapTemplate" class="org.springframework.ldap.core.LdapTemplate">
      <constructor-arg ref="contextSource" />
   </bean>

   <bean id="transactionManager"

 class="org.springframework.ldap.transaction.compensating.manager.ContextSourceTransactionManager">
      <property name="contextSource" ref="contextSource" />
   </bean>

   <bean id="myDataAccessObjectTarget" class="com.example.MyDataAccessObject">
      <property name="ldapTemplate" ref="ldapTemplate" />
   </bean>

   <bean id="myDataAccessObject"

 class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
      <property name="transactionManager" ref="transactionManager" />
      <property name="target" ref="myDataAccessObjectTarget" />
      <property name="transactionAttributes">
         <props>
            <prop key="*">PROPAGATION_REQUIRES_NEW</prop>
         </props>
      </property>
   </bean>
   ...
```

In a real world example you would probably apply the transactions on the service object level rather than the DAO level; the above serves as an example to demonstrate the general idea.

Note

You'll notice that the actual `ContextSource` and DAO instances get ids with a "Target" suffix. The beans you will actually refer to are the Proxies that are created around the targets; `contextSource` and `myDataAccessObject`

6.3. JDBC Transaction Integration

A common use case when working against LDAP is that some of the data is stored in the LDAP tree, but other data is stored in a relational database. In this case, transaction support becomes even more important, since the update of the different resources should be synchronized.

While actual XA transactions is not supported, support is provided to conceptually wrap JDBC and LDAP access within the same transaction using the `ContextSourceAndDataSourceTransactionManager`. A `DataSource` and a `ContextSource` is supplied to the `ContextSourceAndDataSourceTransactionManager`, which will then manage the two transactions, virtually as if they were one. When performing a commit, the LDAP part of the operation will always be performed first, allowing both transactions to be rolled back should the LDAP commit fail. The JDBC part of the transaction is managed exactly as in `DataSourceTransactionManager`, except that nested transactions is not supported.

Note

Once again it should be noted that the provided support is all client side. The wrapped transaction is not an XA transaction. No two-phase as such commit is performed, as the LDAP server will be unable to vote on its outcome. Once again, however, for the majority of cases the supplied support will be sufficient.

6.4. LDAP Compensating Transactions Explained

Spring LDAP manages compensating transactions by making record of the state in the LDAP tree before each modifying operation (`bind`, `unbind`, `rebind`, `modifyAttributes`, and `rename`).

This enables the system to perform compensating operations should the transaction need to be rolled back. In many cases the compensating operation is pretty straightforward. E.g. the compensating rollback operation for a `bind` operation will quite obviously be to unbind the entry. Other operations however require a different, more complicated approach because of some particular characteristics of LDAP databases. Specifically, it is not always possible to get the values of all `Attributes` of an entry, making the above strategy insufficient for e.g. an `unbind` operation.

This is why each modifying operation performed within a Spring LDAP managed transaction is internally split up in four distinct operations - a recording operation, a preparation operation, a commit operation, and a rollback operation. The specifics for each LDAP operation is described in the table below:

Table 6.1.

LDAP Operation	Recording	Preparation	Commit	Rollback
`bind`	Make record of the DN of the entry to bind.	Bind the entry.	No operation.	Unbind the entry using the recorded DN.
`rename`	Make record of the original and target DN.	Rename the entry.	No operation.	Rename the entry back to its original DN.
`unbind`	Make record of the original DN and calculate a temporary DN.	Rename the entry to the temporary location.	Unbind the temporary entry.	Rename the entry from the temporary location back to its original DN.
`rebind`	Make record of the original DN and the new `Attributes`,	Rename the entry to a temporary location.	Bind the new `Attributes` at the original DN, and unbind the	Rename the entry from the temporary

LDAP Operation	Recording	Preparation	Commit	Rollback
	and calculate a temporary DN.		original entry from its temporary location.	location back to its original DN.
`modifyAttributes`	Make record of the DN of the entry to modify and calculate compensating `ModificationItems` for the modifications to be done.	Perform the `modifyAttributes` operation.	No operation.	Perform a `modifyAttributes` operation using the calculated compensating `ModificationItems`.

A more detailed description of the internal workings of the Spring LDAP transaction support is available in the javadocs.

Renaming Strategies

As described in the table above, the transaction management of some operations require the original entry affected by the operation to be temporarily renamed before the actual modification can be made in the commit. The manner in which the temporary DN of the entry is calculated is managed by a `TempEntryRenamingStrategy` supplied to the `ContextSourceTransactionManager`. Two implementations are supplied with Spring LDAP, but if specific behaviour is required a custom implementation can easily be implemented by the user. The provided `TempEntryRenamingStrategy` implementations are:

- `DefaultTempEntryRenamingStrategy` (the default). Adds a suffix to the least significant part of the entry DN. E.g. for the DN `cn=john doe, ou=users`, this strategy would return the temporary DN `cn=john doe_temp, ou=users`. The suffix is configurable using the `tempSuffix` property
- `DifferentSubtreeTempEntryRenamingStrategy`. Takes the least significant part of the DN and appends a subtree DN to this. This makes all temporary entries be placed at a specific location in the LDAP tree. The temporary subtree DN is configured using the `subtreeNode` property. E.g., if `subtreeNode` is `ou=tempEntries` and the original DN of the entry is `cn=john doe, ou=users`, the temporary DN will be `cn=john doe, ou=tempEntries`. Note that the configured subtree node needs to be present in the LDAP tree.

Note

There are some situations where the `DefaultTempEntryRenamingStrategy` will not work. E.g. if your are planning to do recursive deletes you'll need to use `DifferentSubtreeTempEntryRenamingStrategy`. This is because the recursive delete operation actually consists of a depth-first delete of each node in the sub tree individually. Since it is not allowed to rename an entry that has any children, and `DefaultTempEntryRenamingStrategy` would leave each node in the same subtree (with a different name) in stead of actually removing it, this operation would fail. When in doubt, use `DifferentSubtreeTempEntryRenamingStrategy`.

7. Java 5 Support

7.1. SimpleLdapTemplate

As of version 1.3 Spring LDAP includes the spring-ldap-core-tiger.jar distributable, which adds a thin layer of Java 5 functionality on top of Spring LDAP.

The `SimpleLdapTemplate` class adds search and lookup methods that take a `ParameterizedContextMapper`, adding generics support to these methods.

`ParametrizedContextMapper` is a typed version of `ContextMapper`, which simplifies working with searches and lookups:

```
public List<Person> getAllPersons(){
    return simpleLdapTemplate.search("", "(objectclass=person)",
               new ParameterizedContextMapper<Person>() {
                   public Person mapFromContext(Object ctx) {
                       DirContextAdapter adapter = (DirContextAdapter) ctx;
                       Person person = new Person();
                       // Fill the domain object with data from the DirContextAdapter

                       return person;
                   }
               };
}
```

Example 7.1 Using `ParameterizedContextMapper`

8. Configuration

8.1. ContextSource Configuration

There are several properties in `AbstractContextSource` (superclass of `DirContextSource` and `LdapContextSource`) that can be used to modify its behaviour.

LDAP Server URLs

The URL of the LDAP server is specified using the `url` property. The URL should be in the format `ldap://myserver.example.com:389`. For SSL access, use the `ldaps` protocol and the appropriate port, e.g. `ldaps://myserver.example.com:636`

It is possible to configure multiple alternate LDAP servers using the `urls` property. In this case, supply all server urls in a String array to the `urls` property.

Base LDAP path

It is possible to specify the root context for all LDAP operations using the `base` property of `AbstractContextSource`. When a value has been specified to this property, all Distinguished Names supplied to and received from LDAP operations will be relative to the LDAP path supplied. This can significantly simplify working against the LDAP tree; however there are several occasions when you will need to have access to the base path. For more information on this, please refer to Section 8.3, “Obtaining a reference to the base LDAP path”

DirContext Authentication

When `DirContext` instances are created to be used for performing operations on an LDAP server these contexts often need to be authenticated. There are different options for configuring this using Spring LDAP, described in this chapter.

Note

This section refers to authenticating contexts in the core functionality of the `ContextSource` - to construct `DirContext` instances for use by `LdapTemplate`. LDAP is commonly used for the sole purpose of user authentication, and the `ContextSource` may be used for that as well. This process is discussed in Chapter 10, *User Authentication using Spring LDAP*.

Authenticated contexts are created for both read-only and read-write operations by default. You specify `userDn` and `password` of the LDAP user to be used for authentication on the `ContextSource`.

Note

The `userDn` needs to be the full Distinguished Name (DN) of the user from the root of the LDAP tree, regardless of whether a `base` LDAP path has been supplied to the `ContextSource`.

Some LDAP server setups allow anonymous read-only access. If you want to use anonymous Contexts for read-only operations, set the `anonymousReadOnly` property to `true`.

Custom DirContext Authentication Processing

The default authentication mechanism used in Spring LDAP is SIMPLE authentication. This means that in the user DN (as specified to the `userDn` property) and the credentials (as specified to the `password`) are set in the Hashtable sent to the `DirContext` implementation constructor.

There are many occasions when this processing is not sufficient. For instance, LDAP Servers are commonly set up to only accept communication on a secure TLS channel; there might be a need to use the particular LDAP Proxy Auth mechanism, etc.

It is possible to specify an alternative authentication mechanism by supplying a `DirContextAuthenticationStrategy` implementation to the `ContextSource` in the configuration.

TLS

Spring LDAP provides two different configuration options for LDAP servers requiring TLS secure channel communication: `DefaultTlsDirContextAuthenticationStrategy` and `ExternalTlsDirContextAuthenticationStrategy`. Both these implementations will negotiate a TLS channel on the target connection, but they differ in the actual authentication mechanism. Whereas the `DefaultTlsDirContextAuthenticationStrategy` will apply SIMPLE authentication on the secure channel (using the specified `userDn` and `password`), the `ExternalDirContextAuthenticationStrategy` will use EXTERNAL SASL authentication, applying a client certificate configured using system properties for authentication.

Since different LDAP server implementations respond differently to explicit shutdown of the TLS channel (some servers require the connection be shutdown gracefully; others do not support it), the TLS `DirContextAuthenticationStrategy` implementations support specifying the shutdown behavior using the `shutdownTlsGracefully` parameter. If this property is set to `false` (the default), no explicit TLS shutdown will happen; if it is `true`, Spring LDAP will try to shutdown the TLS channel gracefully before closing the target context.

Note

When working with TLS connections you need to make sure that the native LDAP Pooling functionality is turned off. As of release 1.3, the default setting is off. For earlier versions, simply set the `pooled` property to `false`. This is particularly important if `shutdownTlsGracefully` is set to `false`. However, since the TLS channel negotiation process is quite expensive, great performance benefits will be gained by using the Spring LDAP Pooling Support, described in Chapter 9, *Pooling Support*.

Custom Principal and Credentials Management

While the user name (i.e. user DN) and password used for creating an authenticated `Context` are static by default - the ones set on the `ContextSource` on startup will be used throughout the lifetime of the `ContextSource` - there are however several cases in which this is not the desired behaviour. A common scenario is that the principal and credentials of the current user should be used when executing LDAP operations for that user. The default behaviour can be modified by supplying a custom `AuthenticationSource` implementation to the `ContextSource` on startup, instead of explicitly specifying the `userDn` and `password`. The `AuthenticationSource` will be queried by the `ContextSource` for principal and credentials each time an authenticated `Context` is to be created.

If you are using Spring Security you can make sure the principal and credentials of the currently logged in user is used at all times by configuring your `ContextSource` with an instance of the `SpringSecurityAuthenticationSource` shipped with Spring Security.

```
<beans>
   ...
   <bean id="contextSource"
 class="org.springframework.ldap.core.support.LdapContextSource">
      <property name="url" value="ldap://localhost:389" />
      <property name="base" value="dc=example,dc=com" />
      <property name="authenticationSource" ref="springSecurityAuthenticationSource" />
   </bean>

   <bean id="springSecurityAuthenticationSource"
      class="org.springframework.security.ldap.SpringSecurityAuthenticationSource" />
   ...
</beans>
```

Example 8.1 The Spring bean definition for a SpringSecurityAuthenticationSource

Note

We don't specify any `userDn` or `password` to our `ContextSource` when using an `AuthenticationSource` - these properties are needed only when the default behaviour is used.

Note

When using the `SpringSecurityAuthenticationSource` you need to use Spring Security's `LdapAuthenticationProvider` to authenticate the users against LDAP.

Default Authentication

When using `SpringSecurityAuthenticationSource`, authenticated contexts will only be possible to create once the user is logged in using Spring Security. To use default authentication information when no user is logged in, use the `DefaultValuesAuthenticationSourceDecorator`:

```
<beans>
   ...
   <bean id="contextSource"
 class="org.springframework.ldap.core.support.LdapContextSource">
      <property name="url" value="ldap://localhost:389" />
      <property name="base" value="dc=example,dc=com" />
      <property name="authenticationSource" ref="authenticationSource" />
   </bean>

   <bean id="authenticationSource"

 class="org.springframework.ldap.authentication.DefaultValuesAuthenticationSourceDecorator">
      <property name="target" ref="springSecurityAuthenticationSource" />
      <property name="defaultUser" value="cn=myDefaultUser" />
      <property name="defaultPassword" value="pass" />
   </bean>

   <bean id="springSecurityAuthenticationSource"
      class="org.springframework.security.ldap.SpringSecurityAuthenticationSource" />
   ...
</beans>
```

Example 8.2 Configuring a DefaultValuesAuthenticationSourceDecorator

Native Java LDAP Pooling

The internal Java LDAP provider provides some very basic pooling capabilities. This LDAP connection pooling can be turned on/off using the `pooled` flag on `AbstractContextSource`. The default value is `false` (since release 1.3), i.e. the native Java LDAP pooling will be turned on. The configuration of LDAP connection pooling is managed using `System` properties, so this needs to be handled manually, outside of the Spring Context configuration. Details of the native pooling configuration can be found here.

Note

There are several serious deficiencies in the built-in LDAP connection pooling, which is why Spring LDAP provides a more sophisticated approach to LDAP connection pooling, described in Chapter 9, *Pooling Support*. If pooling functionality is required, this is the recommended approach.

Note

Regardless of the pooling configuration, the `ContextSource#getContext(String principal, String credentials)` method will always explicitly *not* use native Java LDAP Pooling, in order for reset passwords to take effect as soon as possible.

Advanced ContextSource Configuration

Alternate ContextFactory

It is possible to configure the `ContextFactory` that the `ContextSource` is to use when creating Contexts using the `contextFactory` property. The default value is `com.sun.jndi.ldap.LdapCtxFactory`.

Custom DirObjectFactory

As described in Chapter 3, *Simpler Attribute Access and Manipulation with DirContextAdapter*, a `DirObjectFactory` can be used to translate the `Attributes` of found Contexts to a more useful `DirContext` implementation. This can be configured using the `dirObjectFactory` property. You can use this property if you have your own, custom `DirObjectFactory` implementation.

The default value is `DefaultDirObjectFactory`.

Custom DirContext Environment Properties

In some cases the user might want to specify additional environment setup properties in addition to the ones directly configurable from `AbstractContextSource`. Such properties should be set in a `Map` and supplied to the `baseEnvironmentProperties` property.

8.2. LdapTemplate Configuration

Ignoring PartialResultExceptions

Some Active Directory (AD) servers are unable to automatically following referrals, which often leads to a `PartialResultException` being thrown in searches. You can specify that `PartialResultException` is to be ignored by setting the `ignorePartialResultException` property to `true`.

Note

This causes all referrals to be ignored, and no notice will be given that a `PartialResultException` has been encountered. There is currently no way of manually following referrals using LdapTemplate.

8.3. Obtaining a reference to the base LDAP path

As described above, a base LDAP path may be supplied to the `ContextSource`, specifying the root in the LDAP tree to which all operations will be relative. This means that you will only be working with relative distinguished names throughout your system, which is typically rather handy. There are however some cases in which you will need to have access to the base path in order to be able to construct full DNs, relative to the actual root of the LDAP tree. One example would be when working with LDAP groups (e.g. `groupOfNames` objectclass), in which case each group member attribute value will need to be the full DN of the referenced member.

For that reason, Spring LDAP has a mechanism by which any Spring controlled bean may be supplied the base path on startup. For beans to be notified of the base path, two things need to be in place: First of all, the bean that wants the base path reference needs to implement the `BaseLdapPathAware` interface. Secondly, a `BaseLdapPathBeanPostProcessor` needs to be defined in the application context

```
package com.example.service;

public class PersonService implements PersonService, BaseLdapPathAware {
   ...
   private DistinguishedName basePath;

   public void setBaseLdapPath(DistinguishedName basePath) {
      this.basePath = basePath;
   }
   ...
   private DistinguishedName getFullPersonDn(Person person) {
      return new DistinguishedName(basePath).append(person.getDn());
   }
   ...
}
```

Example 8.3 Implementing `BaseLdapPathAware`

```
<beans>
   ...
   <bean id="contextSource"
 class="org.springframework.ldap.core.support.LdapContextSource">
      <property name="url" value="ldap://localhost:389" />
      <property name="base" value="dc=example,dc=com" />
      <property name="authenticationSource" ref="authenticationSource" />
   </bean>
   ...
   <bean class="org.springframework.ldap.core.support.BaseLdapPathBeanPostProcessor" />
</beans>
```

Example 8.4 Specifying a `BaseLdapPathBeanPostProcessor` in your `ApplicationContext`

The default behaviour of the `BaseLdapPathBeanPostProcessor` is to use the base path of the single defined `BaseLdapPathSource` (`AbstractContextSource`)in the `ApplicationContext`. If more than one `BaseLdapPathSource` is defined, you will need to specify which one to use with the `baseLdapPathSourceName` property.

9. Pooling Support

9.1. Introduction

Pooling LDAP connections helps mitigate the overhead of creating a new LDAP connection for each LDAP interaction. While Java LDAP pooling support exists it is limited in its configuration options and features, such as connection validation and pool maintenance. Spring LDAP provides support for detailed pool configuration on a per- `ContextSource` basis.

Pooling support is provided by `PoolingContextSource` which can wrap any `ContextSource` and pool both read-only and read-write `DirContext` objects. Jakarta Commons-Pool is used to provide the underlying pool implementation.

9.2. DirContext Validation

Validation of pooled connections is the primary motivation for using a custom pooling library versus the JDK provided LDAP pooling functionality. Validation allows pooled `DirContext` connections to be checked to ensure they are still properly connected and configured when checking them out of the pool, in to the pool or while idle in the pool

The `DirContextValidator` interface is used by the `PoolingContextSource` for validation and `DefaultDirContextValidator` is provided as the default validation implementation. `DefaultDirContextValidator` does a `DirContext.search(String, String, SearchControls)` , with an empty name, a filter of `"objectclass=*"` and `SearchControls` set to limit a single result with the only the objectclass attribute and a 500ms timeout. If the returned `NamingEnumeration` has results the `DirContext` passes validation, if no results are returned or an exception is thrown the `DirContext` fails validation. The `DefaultDirContextValidator` should work with no configuration changes on most LDAP servers and provide the fastest way to validate the `DirContext` .

9.3. Pool Properties

The following properties are available on the `PoolingContextSource` for configuration of the DirContext pool. The `contextSource` property must be set and the `dirContextValidator` property must be set if validation is enabled, all other properties are optional.

Table 9.1. Pooling Configuration Properties

Parameter	Default	Description
`contextSource`	`null`	The `ContextSource` implementation to get `DirContext` s from to populate the pool.
`dirContextValidator`	`null`	The `DirContextValidator` implementation to use when validating connections. This is required if `testOnBorrow` , `testOnReturn` , or

Parameter	Default	Description
		`testWhileIdle` options are set to `true` .
`maxActive`	`8`	The maximum number of active connections of each type (read-only|read-write) that can be allocated from this pool at the same time, or non-positive for no limit.
`maxTotal`	`-1`	The overall maximum number of active connections (for all types) that can be allocated from this pool at the same time, or non-positive for no limit.
`maxIdle`	`8`	The maximum number of active connections of each type (read-only|read-write) that can remain idle in the pool, without extra ones being released, or non-positive for no limit.
`minIdle`	`0`	The minimum number of active connections of each type (read-only|read-write) that can remain idle in the pool, without extra ones being created, or zero to create none.
`maxWait`	`-1`	The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception, or non-positive to wait indefinitely.
`whenExhaustedAction`	`1` (BLOCK)	Specifies the behaviour when the pool is exhausted. • The FAIL (`0`) option will throw a `NoSuchElementException` when the pool is exhausted. • The BLOCK (`1`) option will wait until a new object is available. If `maxWait` is positive a

Parameter	Default	Description
		`NoSuchElementException` is thrown if no new object is available after the `maxWait` time expires. • The GROW (2) option will create and return a new object (essentially making `maxActive` meaningless).
`testOnBorrow`	`false`	The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and an attempt to borrow another will be made.
`testOnReturn`	`false`	The indication of whether objects will be validated before being returned to the pool.
`testWhileIdle`	`false`	The indication of whether objects will be validated by the idle object evictor (if any). If an object fails to validate, it will be dropped from the pool.
`timeBetweenEvictionRunsMillis`	`-1`	The number of milliseconds to sleep between runs of the idle object evictor thread. When non-positive, no idle object evictor thread will be run.
`numTestsPerEvictionRun`	`3`	The number of objects to examine during each run of the idle object evictor thread (if any).
`minEvictableIdleTimeMillis`	`1000 * 60 * 30`	The minimum amount of time an object may sit idle in the pool before it is eligible for eviction by the idle object evictor (if any).

9.4. Configuration

Configuring pooling should look very familiar if you're used to Jakarta Commons-Pool or Commons-DBCP. You will first create a normal `ContextSource` then wrap it in a `PoolingContextSource` .

```
<beans>
   ...
   <bean id="contextSource"
 class="org.springframework.ldap.pool.factory.PoolingContextSource">
      <property name="contextSource" ref="contextSourceTarget" />
   </bean>

   <bean id="contextSourceTarget"
 class="org.springframework.ldap.core.support.LdapContextSource">
      <property name="url" value="ldap://localhost:389" />
      <property name="base" value="dc=example,dc=com" />
      <property name="userDn" value="cn=Manager" />
      <property name="password" value="secret" />
      <property name="pooled" value="false"/>
   </bean>
   ...
</beans>
```

In a real world example you would probably configure the pool options and enable connection validation; the above serves as an example to demonstrate the general idea.

Note

Ensure that the `pooled` property is set to `false` on any `ContextSource` that will be wrapped in a `PoolingContextSource` . The `PoolingContextSource` must be able to create new connections when needed and if `pooled` is set to `true` that may not be possible.

Note

You'll notice that the actual `ContextSource` gets an id with a "Target" suffix. The bean you will actually refer to is the `PoolingContextSource` that wraps the target `contextSource`

Validation Configuration

Adding validation and a few pool configuration tweaks to the above example is straight forward. Inject a `DirContextValidator` and set when validation should occur and the pool is ready to go.

```
<beans>
   ...
   <bean id="contextSource"
 class="org.springframework.ldap.pool.factory.PoolingContextSource">
      <property name="contextSource" ref="contextSourceTarget" />
      <property name="dirContextValidator" ref="dirContextValidator" />
      <property name="testOnBorrow" value="true" />
      <property name="testWhileIdle" value="true" />
   </bean>

   <bean id="dirContextValidator"
         class="org.springframework.ldap.pool.validation.DefaultDirContextValidator" />

   <bean id="contextSourceTarget"
 class="org.springframework.ldap.core.support.LdapContextSource">
      <property name="url" value="ldap://localhost:389" />
      <property name="base" value="dc=example,dc=com" />
      <property name="userDn" value="cn=Manager" />
      <property name="password" value="secret" />
      <property name="pooled" value="false"/>
   </bean>
   ...
</beans>
```

The above example will test each `DirContext` before it is passed to the client application and test `DirContext` s that have been sitting idle in the pool.

9.5. Known Issues

Custom Authentication

The `PoolingContextSource` assumes that all `DirContext` objects retrieved from `ContextSource.getReadOnlyContext()` will have the same environment and likewise that all `DirContext` objects retrieved from `ContextSource.getReadWriteContext()` will have the same environment. This means that wrapping a `LdapContextSource` configured with an `AuthenticationSource` in a `PoolingContextSource` will not function as expected. The pool would be populated using the credentials of the first user and unless new connections were needed subsequent context requests would not be filled for the user specified by the `AuthenticationSource` for the requesting thread.

10. User Authentication using Spring LDAP

10.1. Basic Authentication

While the core functionality of the `ContextSource` is to provide `DirContext` instances for use by `LdapTemplate`, it may also be used for authenticating users against an LDAP server. The `getContext(principal, credentials)` method of `ContextSource` will do exactly that; construct a `DirContext` instance according to the `ContextSource` configuration, authenticating the context using the supplied principal and credentials. A custom authenticate method could look like this:

```
public boolean authenticate(String userDn, String credentials) {
  DirContext ctx = null;
  try {
    ctx = contextSource.getContext(userDn, credentials);
    return true;
  } catch (Exception e) {
    // Context creation failed - authentication did not succeed
    logger.error("Login failed", e);
    return false;
  } finally {
    // It is imperative that the created DirContext instance is always closed
    LdapUtils.closeContext(ctx);
  }
}
```

The userDn supplied to the `authenticate` method needs to be the full DN of the user to authenticate (regardless of the `base` setting on the `ContextSource`). You will typically need to perform an LDAP search based on e.g. the user name to get this DN:

```
private String getDnForUser(String uid) {
  Filter f = new EqualsFilter("uid", uid);
  List result = ldapTemplate.search(DistinguishedName.EMPTY_PATH, f.toString(),
      new AbstractContextMapper() {
    protected Object doMapFromContext(DirContextOperations ctx) {
      return ctx.getNameInNamespace();
    }
  });

  if(result.size() != 1) {
    throw new RuntimeException("User not found or not unique");
  }

  return (String)result.get(0);
}
```

There are some drawbacks to this approach. The user is forced to concern herself with the DN of the user, she can only search for the user's uid, and the search always starts at the root of the tree (the empty path). A more flexible method would let the user specify the search base, the search filter, and the credentials. Spring LDAP 1.3.0 introduced new authenticate methods in LdapTemplate that provide this functionality:

- `boolean authenticate(Name base, String filter, String password);`
- `boolean authenticate(String base, String filter, String password);`

Using one of these methods, authentication becomes as simple as this:

```
boolean authenticated = ldapTemplate.authenticate("", "(uid=john.doe)", "secret");
```

Example 10.1 Authenticating a user using Spring LDAP.

Note

As described in below, some setups may require additional operations to be performed in order for actual authentication to occur. See Section 10.2, “Performing Operations on the Authenticated Context” for details.

Tip

Don't write your own custom authenticate methods. Use the ones provided in Spring LDAP 1.3.x.

10.2. Performing Operations on the Authenticated Context

Some authentication schemes and LDAP servers require some operation to be performed on the created `DirContext` instance for the actual authentication to occur. You should test and make sure how your server setup and authentication schemes behave; failure to do so might result in that users will be admitted into your system regardless of the DN/credentials supplied. This is a naïve implementation of an authenticate method where a hard-coded `lookup` operation is performed on the authenticated context:

```
public boolean authenticate(String userDn, String credentials) {
  DirContext ctx = null;
  try {
    ctx = contextSource.getContext(userDn, credentials);
    // Take care here - if a base was specified on the ContextSource
    // that needs to be removed from the user DN for the lookup to succeed.
    ctx.lookup(userDn);
    return true;
  } catch (Exception e) {
    // Context creation failed - authentication did not succeed
    logger.error("Login failed", e);
    return false;
  } finally {
    // It is imperative that the created DirContext instance is always closed
    LdapUtils.closeContext(ctx);
  }
}
```

It would be better if the operation could be provided as an implementation of a callback interface, thus not limiting the operation to always be a `lookup`. Spring LDAP 1.3.0 introduced the callback interface `AuthenticatedLdapEntryContextCallback` and a few corresponding `authenticate` methods:

- `boolean authenticate(Name base, String filter, String password, AuthenticatedLdapEntryContextCallback callback);`
- `boolean authenticate(String base, String filter, String password, AuthenticatedLdapEntryContextCallback callback);`

This opens up for any operation to be performed on the authenticated context:

```
AuthenticatedLdapEntryContextCallback contextCallback = new
 AuthenticatedLdapEntryContextCallback() {
  public void executeWithContext(DirContext ctx, LdapEntryIdentification
 ldapEntryIdentification) {
    try {
      ctx.lookup(ldapEntryIdentification.getRelativeDn());
    }
    catch (NamingException e) {
      throw new RuntimeException("Failed to lookup " +
 ldapEntryIdentification.getRelativeDn(), e);
    }
  }
};

ldapTemplate.authenticate("", "(uid=john.doe)", "secret", contextCallback));
```

Example 10.2 Performing an LDAP operation on the authenticated context using Spring LDAP.

10.3. Retrieving the Authentication Exception

So far, the methods have only been able to tell the user whether or not the authentication succeeded. There has been no way of retrieving the actual exception. Spring LDAP 1.3.1 introduced the `AuthenticationErrorCallback` and a few more `authenticate` methods:

- `boolean authenticate(Name base, String filter, String password, AuthenticationErrorCallback errorCallback);`
- `boolean authenticate(String base, String filter, String password, AuthenticationErrorCallback errorCallback);`
- `boolean authenticate(Name base, String filter, String password, AuthenticatedLdapEntryContextCallback callback, AuthenticationErrorCallback errorCallback);`
- `boolean authenticate(String base, String filter, String password, AuthenticatedLdapEntryContextCallback callback, AuthenticationErrorCallback errorCallback);`

A convenient collecting implementation of the error callback interface is also provided:

```
public final class CollectingAuthenticationErrorCallback implements
 AuthenticationErrorCallback {
  private Exception error;

  public void execute(Exception e) {
    this.error = e;
  }

  public Exception getError() {
    return error;
  }
}
```

The code needed for authenticating a user and retrieving the authentication exception in case of an error boils down to this:

```
import org.springframework.ldap.core.support.CollectingAuthenticationErrorCallback;
...
CollectingAuthenticationErrorCallback errorCallback = new
 CollectingAuthenticationErrorCallback();
boolean result = ldapTemplate.authenticate("", filter.toString(), "invalidpassword",
 errorCallback);
if (!result) {
  Exception error = errorCallback.getError();
  // error is likely of type org.springframework.ldap.AuthenticationException
}
```

Example 10.3 Authenticating a user and retrieving the authentication exception.

10.4. Use Spring Security

While the approach above may be sufficient for simple authentication scenarios, requirements in this area commonly expand rapidly. There is a multitude of aspects that apply, including authentication, authorization, web integration, user context management, etc. If you suspect that the requirements might expand beyond just simple authentication, you should definitely consider using Spring Security for your security purposes instead. It is a full-blown, mature security framework addressing the above aspects as well as several others.

11. LDIF Parsing

11.1 Introduction

LDAP Directory Interchange Format (LDIF) files are the standard medium for describing directory data in a flat file format. The most common uses of this format include information transfer and archival. However, the standard also defines a way to describe modifications to stored data in a flat file format. LDIFs of this later type are typically referred to as *changetype* or *modify* LDIFs.

The org.springframework.ldap.ldif package provides classes needed to parse LDIF files and deserialize them into tangible objects. The LdifParser is the main class of the org.springframework.ldap.ldif package and is capable of parsing files that are RFC 2849 compliant. This class reads lines from a resource and assembles them into an LdapAttributes object. The LdifParser currently ignores *changetype* LDIF entries as their usefulness in the context of an application has yet to be determined.

11.2 Object Representation

Two classes in the org.springframework.ldap.core package provide the means to represent an LDIF in code:

- LdapAttribute - Extends javax.naming.directory.BasicAttribute adding support for LDIF options as defined in RFC2849.
- LdapAttributes - Extends javax.naming.directory.BasicAttributes adding specialized support for DNs.

LdapAttribute objects represent options as a Set<String>. The DN support added to the LdapAttributes object employs the org.springframework.ldap.core.DistinguishedName class.

11.3 The Parser

The Parser interface provides the foundation for operation and employs three supporting policy definitions:

- SeparatorPolicy - establishes the mechanism by which lines are assembled into attributes.
- AttributeValidationPolicy - ensures that attributes are correctly structured prior to parsing.
- Specification - provides a mechanism by which object structure can be validated after assembly.

The default implementations of these interfaces are the org.springframework.ldap.ldif.parser.LdifParser, the org.springframework.ldap.ldif.support.SeparatorPolicy, and the org.springframework.ldap.ldif.support.DefaultAttributeValidationPolicy, and the org.springframework.ldap.schema.DefaultSchemaSpecification respectively. Together, these 4 classes parse a resource line by line and translate the data into LdapAttributes objects.

The SeparatorPolicy determines how individual lines read from the source file should be interpreted as the LDIF specification allows attributes to span multiple lines. The default policy assess lines in the context of the order in which they were read to determine the nature of the line in consideration. *control* attributes and *changetype* records are ignored.

The DefaultAttributeValidationPolicy uses REGEX expressions to ensure each attribute conforms to a valid attribute format according to RFC 2849 once parsed. If an attribute fails validation, an InvalidAttributeFormatException is logged and the record is skipped (the parser returns null).

11.4 Schema Validation

A mechanism for validating parsed objects against a schema and is available via the Specification interface in the org.springframework.ldap.schema package. The DefaultSchemaSpecification does not do any validation and is available for instances where records are known to be valid and not required to be checked. This option saves the performance penalty that validation imposes. The BasicSchemaSpecification applies basic checks such as ensuring DN and object class declarations have been provided. Currently, validation against an actual schema requires implementation of the Specification interface.

11.5 Spring Batch Integration

While the LdifParser can be employed by any application that requires parsing of LDIF files, Spring offers a batch processing framework that offers many file processing utilities for parsing delimited files such as CSV. The org.springframework.ldap.ldif.batch package offers the classes necessary for using the LdifParser as a valid configuration option in the Spring Batch framework.

There are 5 classes in this package which offer three basic use cases:

- Use Case 1: Read LDIF records from a file and return an LdapAttributes object.
- Use Case 2: Read LDIF records from a file and map records to Java objects (POJOs).
- Use Case 3: Write LDIF records to a file.

The first use case is accomplished with the LdifReader. This class extends Spring Batch's AbstractItemCountingItemSteamItemReader and implements its ResourceAwareItemReaderItemStream. It fits naturally into the framework and can be used to read LdapAttributes objects from a file.

The MappingLdifReader can be used to map LDIF objects directly to any POJO. This class requires an implementation of the RecordMapper interface be provided. This implementation should implement the logic for mapping objects to POJOs.

The RecordCallbackHandler can be implemented and provided to either reader. This handler can be used to operate on skipped records. Consult the Spring Batch documentation for more information.

The last member of this package, the LdifAggregator, can be used to write LDIF records to a file. This class simply invokes the toString() method of the LdapAttributes object.

12. Object-Directory Mapping (ODM)

12.1. Introduction

Relational mapping frameworks like Hibernate and JPA have offered developers the ability to use annotations to map database tables to Java objects for some time. The Spring Framework LDAP project now offers the same ability with respect to directories through the use of the `org.springframework.ldap.odm` package (sometimes abbreviated as `o.s.l.odm`).

12.2. OdmManager

The `org.springframework.ldap.odm.OdmManager` interface, and its implementation, is the central class in the ODM package. The `OdmManager` orchestrates the process of reading objects from the directory and mapping the data to annotated Java object classes. This interface provides access to the underlying directory instance through the following methods:

- `<T> T read(Class<T> clazz, Name dn)`
- `void create(Object entry)`
- `void update(Object entry)`
- `void delete(Object entry)`
- `<T> List<T> findAll(Class<T> clazz, Name base, SearchControls searchControls)`
- `<T> List<T> search(Class<T> clazz, Name base, String filter, SearchControls searchControls)`

A reference to an implementation of this interface can be obtained through the `org.springframework.ldap.odm.core.impl.OdmManagerImplFactoryBean`. A basic configuration of this factory would be as follows:

```
<beans>
   ...
   <bean id="odmManager"
         class="org.springframework.ldap.odm.core.impl.OdmManagerImplFactoryBean">
      <property name="converterManager" ref="converterManager" />
      <property name="contextSource" ref="contextSource" />
      <property name="managedClasses">
         <set>
            <value>com.example.dao.SimplePerson</value>
         </set>
      </property>
   </bean>
   ...
</beans>
```

Example 12.1 Configuring the OdmManager Factory

The factory requires the list of entity classes to be managed by the `OdmManager` to be explicitly declared. These classes should be properly annotated as defined in the next section. The `converterManager` referenced in the above definition is described in Section 12.4, “Type Conversion”.

12.3. Annotations

Entity classes managed by the `OdmManager` are required to be annotated with the annotations in the `org.springframework.ldap.odm.annotations` package. The available annotations are:

- `@Entry` - Class level annotation indicating the `objectClass` definitions to which the entity maps. *(required)*
- `@Id` - Indicates the entity DN; the field declaring this attribute must be a derivative of the `javax.naming.Name` class. *(required)*
- `@Attribute` - Indicates the mapping of a directory attribute to the object class field.
- `@Transient` - Indicates the field is not persistent and should be ignored by the `OdmManager`.

The `@Entry` and `@Id` attributes are required to be declared on managed classes. `@Entry` is used to specify which object classes the entity maps too. All object classes for which fields are mapped are required to be declared. Also, in order for a directory entry to be considered a match to the managed entity, all object classes declared by the directory entry must match be declared by in the `@Entry` annotation.

The `@Id` annotation is used to map the distinguished name of the entry to a field. The field must be an instance of `javax.naming.Name` or a subclass of it.

The `@Attribute` annotation is used to map object class fields to entity fields. `@Attribute` is required to declare the name of the object class property to which the field maps and may optionally declare the syntax OID of the LDAP attribute, to guarantee exact matching. `@Attribute` also provides the type declaration which allows you to indicate whether the attribute is regarded as binary based or string based by the LDAP JNDI provider.

The `@Transient` annotation is used to indicate the field should be ignored by the `OdmManager` and not mapped to an underlying LDAP property.

12.4. Type Conversion

The `OdmManager` relies on the `org.springframework.ldap.odm.typeconversion` package to convert LDAP attributes to Java fields. The main interface in this class is the `org.springframework.ldap.odm.typeconversion.ConverterManager`. The default `ConverterManager` implementation uses the following algorithm when parsing objects to convert fields:

1. Try to find and use a `Converter` registered for the `fromClass`, `syntax` and `toClass` and use it.
2. If this fails, then if the `toClass` `isAssignableFrom` the `fromClass` then just assign it.
3. If this fails try to find and use a `Converter` registered for the `fromClass` and the `toClass` ignoring the syntax.
4. If this fails then throw a `ConverterException`.

Implementations of the `ConverterManager` interface can be obtained from the `o.s.l.odm.typeconversion.impl.ConvertManagerFactoryBean`. The factory bean requires converter configurations to be declared in the bean configuration.

The converterConfig property accepts a set of `ConverterConfig` classes, each one defining some conversion logic. A converter config is an instance of `o.s.l.odm.typeconversion.impl.ConverterManagerFactoryBean.ConverterConfig`. The config defines a set of source classes, the set of target classes, and an implementation of the `org.springframework.ldap.odm.typeconversion.impl.Converter` interface which provides the logic to convert from the `fromClass` to the `toClass`. A sample configuration is provided in the following example:

```
<bean id="fromStringConverter"

 class="org.springframework.ldap.odm.typeconversion.impl.converters.FromStringConverter" /
>
<bean id="toStringConverter"
   class="org.springframework.ldap.odm.typeconversion.impl.converters.ToStringConverter" /
>
<bean id="converterManager"
   class="org.springframework.ldap.odm.typeconversion.impl.ConverterManagerFactoryBean">
   <property name="converterConfig">
      <set>
      <bean class="org.springframework.ldap.odm.\
      typeconversion.impl.ConverterManagerFactoryBean$ConverterConfig">
            <property name="fromClasses">
               <set>
                  <value>java.lang.String</value>
               </set>
            </property>
            <property name="toClasses">
               <set>
                  <value>java.lang.Byte</value>
                  <value>java.lang.Short</value>
                  <value>java.lang.Integer</value>
                  <value>java.lang.Long</value>
                  <value>java.lang.Float</value>
                  <value>java.lang.Double</value>
                  <value>java.lang.Boolean</value>
               </set>
            </property>
            <property name="converter" ref="fromStringConverter" />
         </bean>
         <bean class="org.springframework.ldap.odm.\
  typeconversion.impl.ConverterManagerFactoryBean$ConverterConfig">
            <property name="fromClasses">
               <set>
                  <value>java.lang.Byte</value>
                  <value>java.lang.Short</value>
                  <value>java.lang.Integer</value>
                  <value>java.lang.Long</value>
                  <value>java.lang.Float</value>
                  <value>java.lang.Double</value>
                  <value>java.lang.Boolean</value>
               </set>
            </property>
            <property name="toClasses">
               <set>
                  <value>java.lang.String</value>
               </set>
            </property>
            <property name="converter" ref="toStringConverter" />
         </bean>
      </set>
   </property>
</bean>
```

Example 12.2 Configuring the Converter Manager Factory

12.5. Execution

After all components are configured, directory interaction can be achieved through a reference to the `OdmManager`, as shown in this example:

```
public class App {
   private static Log log = LogFactory.getLog(App.class);
   private static final SearchControls searchControls =
      new SearchControls(SearchControls.SUBTREE_SCOPE, 100, 10000, null, true, false);
   public static void main( String[] args ) {
      try {
         ApplicationContext context = new
 ClassPathXmlApplicationContext("applicationContext.xml");
         OdmManager manager = (OdmManager) context.getBean("odmManager");
  List<SimplePerson> people = manager.search(SimplePerson.class,
     new DistinguishedName("dc=example,dc=com"), "uid=*", searchControls);
         log.info("People found: " + people.size());
         for (SimplePerson person : people) {
            log.info( person );
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Example 12.3 Execution

13. Utilities

13.1. Incremental Retrieval of Multi-Valued Attributes

When there are a very large number of attribute values (>1500) for a specific attribute, Active Directory will typically refuse to return all these values at once. Instead the attribute values will be returned according to the Incremental Retrieval of Multi-valued Properties method. This requires the calling part to inspect the returned attribute for specific markers and, if necessary, make additional lookup requests until all values are found.

Spring LDAP's `org.springframework.ldap.core.support.DefaultIncrementalAttributesMapper` helps working with this kind of attributes, as follows:

```
Attributes attrs = DefaultIncrementalAttributeMapper.lookupAttributes(ldapTemplate, theDn,
 new Object[]{"oneAttribute", "anotherAttribute"});
```

This will parse any returned attribute range markers and make repeated requests as necessary until all values for all requested attributes have been retrieved.