Spring LDAP Reference

Mattias Hellborg Arthursson, Ulrik Sandberg, Eric Dalquist, Keith Barlow, Rob Winch – Version 2.0.2.RELEASE

Table of Contents

Spring LDAP makes it easier to build Spring-based applications that use the Lightweight Directory Access Protocol.

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 core component of 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 is designed to simplify LDAP programming in Java. Some of the features provided by the library are:

- JdbcTemplate-style template simplifications to LDAP programming.

- JPA/Hibernate-style annotation-based object/directory mapping.
- Spring Data repository support, including support for QueryDSL.
- Utilities to simplify building LDAP queries and distinguished names.
- Proper LDAP connection pooling.
- Client-side LDAP compensating transaction support.

1.2. Traditional Java LDAP v/s LdapTemplate

Consider 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.

Working against an LDAP database with JNDI, 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. Note the code marked **bold** - this is the code that actually performs tasks related to the business purpose of the method - the rest is just plumbing:

```
package com.example.repository;

public class TraditionalPersonRepoImpl implements PersonRepo {
   public List<String> 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);
      }

      List<String> list = new LinkedList<String>();
      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 = attr.get().toString();
            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.repo;
import static org.springframework.ldap.query.LdapQueryBuilder.query;

public class PersonRepoImpl implements PersonRepo {
    private LdapTemplate ldapTemplate;

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

    public List<String> getAllPersonNames() {
        return ldapTemplate.search(
            query().where("objectclass").is("person"),
            new AttributesMapper<String>() {
                public String mapFromAttributes(Attributes attrs)
                    throws NamingException {
                    return attrs.get("cn").get().toString();
                }
            });
    }
}
```

The amount of boilerplate code is significantly less than in the traditional example. The `LdapTemplate` search method makes sure a `DirContext` instance is created, 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. It also makes sure that the `NamingEnumeration` and `DirContext` are properly closed and takes care of any exceptions that might happen.

Naturally — this being a Spring Framework sub-project — we will use Spring to configure our application>

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ldap="http://www.springframework.org/schema/ldap"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/ldap http://www.springframework.org/schema/ldap/spring-ldap.xsd">

    <ldap:context-source
           url="ldap://localhost:389"
           base="dc=example,dc=com"
           username="cn=Manager"
           password="secret" />

    <ldap:ldap-template id="ldapTemplate" />

    <bean id="personRepo" class="com.example.repo.PersonRepoImpl">
        <property name="ldapTemplate" ref="ldapTemplate" />
    </bean>
</beans>
```

In order to use the custom XML namespace for configuring the Spring LDAP components you need to include references to this namespace in your XML declaration as in the example above.

1.3. What's new in 2.0?

While quite significant modernizations have been made to the Spring LDAP API in version 2.0, great care has been taken to ensure backward compatibility as far as possible. Code that works with Spring LDAP 1.3.x should with very few exceptions still compile and run using the 2.0 libraries without any modifications whatsoever.

The exception is a small number of classes that have been moved to new packages in order to make a couple of important refactorings

possible. The moved classes are typically not part of the intended public API, and the migration procedure should be very smooth - wherever a Spring LDAP class cannot be found after upgrade, just organize the imports in your IDE.

You will probably encounter some deprecation warnings though, and there are also a lot of other API improvements. The recommendation for getting as much as possible out of the 2.0 version is to move away from the deprecated classes and methods and migrate to the new, improved API utilities.

Below is a list of the most important changes in Spring LDAP 2.0.

- Java 6 is now required by Spring LDAP. Spring versions starting at 2.0 and up are still supported.
- The central API has been updated with Java 5+ features such as generics and varargs. As a consequence, the entire `spring-ldap-tiger` module has been deprecated and users are encouraged to migrate to use the core Spring LDAP classes. The parameterization of the core interfaces will cause lots of compilation warnings on existing code, and users of the API are encouraged to take appropriate action to get rid of these warnings.
- The ODM (Object-Directory Mapping) functionality has been moved to core and there are new methods in `LdapOperations`/`LdapTemplate` that use this automatic translation to/from ODM-annotated classes. See Object-Directory Mapping (ODM) for more information.
- A custom XML namespace is now (finally) provided to simplify configuration of Spring LDAP. See Configuration for more information.
- Spring Data Repository and QueryDSL support is now provided in Spring LDAP. See Spring LDAP Repositories for more information.
- `Name` instances as attribute values are now handled properly with regards to distinguished name equality in `DirContextAdapter` and ODM. See DirContextAdapter and Distinguished Names as Attribute Values. and ODM and Distinguished Names as Attribute Values. for more information.
- `DistinguishedName` and associated classes have been deprecated in favor of standard Java `LdapName`. See Dynamically Building Distinguished Names for information on how the library helps working with `LdapNames`.
- Fluent LDAP query building support has been added. This makes for a more pleasant programming experience when working with LDAP searches in Spring LDAP. See Building LDAP Queries and Advanced LDAP Queries for more information about the LDAP query builder support.
- The old `authenticate` methods in `LdapTemplate` have been deprecated in favor of a couple of new `authenticate` methods that work with `LdapQuery` objects and *throwexceptions* on authentication failure, making it easier for the user to find out what caused an authentication attempt to fail.
- The samples have been polished and updated to make use of the features in 2.0. Quite a bit of effort has been put into providing a useful example of an LDAP user management application.

1.4. 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)
- *spring-data-commons* (base infrastructure for repository suppport, etc.)
- *slf4j* (a simple logging facade, 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)

- *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)

1.5. Getting Started

The samples provide some useful examples on how to use Spring LDAP for common usecases.

1.6. Support

The community support forum is located at http://forum.spring.io/forum/spring-projects/data/ldap, and the project web page is http://spring.io/spring-ldap/.

1.7. Acknowledgements

The initial effort when starting the Spring LDAP project was sponsored by Jayway. Current maintenance of the project is funded by Pivotal

Thanks to Structure101 for providing an open source license that has come in handy for keeping the project structure in check.

---

2. Basic Usage

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.

AttributesMapper that returns a single attribute

```
package com.example.repo;
import static org.springframework.ldap.query.LdapQueryBuilder.query;

public class PersonRepoImpl implements PersonRepo {
   private LdapTemplate ldapTemplate;

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

   public List<String> getAllPersonNames() {
      return ldapTemplate.search(
         query().where("objectclass").is("person"),
         new AttributesMapper<String>() {
            public String mapFromAttributes(Attributes attrs)
               throws NamingException {
               return (String) attrs.get("cn").get();
            }
         });
   }
}
```

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:

AttributesMapper that returns a Person object

```
package com.example.repo;
import static org.springframework.ldap.query.LdapQueryBuilder.query;

public class PersonRepoImpl implements PersonRepo {
   private LdapTemplate ldapTemplate;
   ...
   private class PersonAttributesMapper implements AttributesMapper<Person> {
      public Person 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<Person> getAllPersons() {
        return ldapTemplate.search(query()
            .where("objectclass").is("person"), new PersonAttributesMapper());
    }
}
```

Entries in LDAP are uniquely identified by their distinguished name (DN). If you have the DN of 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 for a `Person` object:

A lookup resulting in a Person object

```
package com.example.repo;

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

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 LDAP Queries

LDAP searches involve a number of parameters, e.g.:

- Base LDAP path - where in the LDAP tree should the search start.
- Search scope - how deep in the LDAP tree should the search go.
- Attributes to return
- Search filter - The criteria to use when selecting elements within scope.

Spring LDAP provides an `LdapQueryBuilder` with a fluent API for building LDAP Queries.

Let's say that we want to perform a search starting at the base DN `dc=261consulting,dc=com`, limiting the returned attributes to "cn" and "sn", with the 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 `LdapQueryBuilder`:

Building a search filter dynamically

```
package com.example.repo;
import static org.springframework.ldap.query.LdapQueryBuilder.query;

public class PersonRepoImpl implements PersonRepo {
    private LdapTemplate ldapTemplate;
    ...
    public List<String> getPersonNamesByLastName(String lastName) {

        LdapQuery query = query()
            .base("dc=261consulting,dc=com")
            .attributes("cn", "sn")
            .where("objectclass").is("person")
            .and("sn").is(lastName);

        return ldapTemplate.search(query,
            new AttributesMapper<String>() {
                public String mapFromAttributes(Attributes attrs)
```

```
            throws NamingException {

            return attrs.get("cn").get();
        }
    });
  }
}
```

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

There are many overloaded methods in `LdapTemplate` for performing LDAP searches. This is in order to accommodate for as many different use cases and programming style preferences as possible. For the vast majority of use cases the ones that take an `LdapQuery` as input will be the recommended methods to use.

The `AttributesMapper` is just one of the available callback interfaces to use when handling search and lookup data. See Simplifying Attribute Access and Manipulation with DirContextAdapter for alternatives.

For more information on the `LdapQueryBuilder` see Advanced LDAP Queries.

2.3. Dynamically Building Distinguished Names

The standard Java implementation of Distinguished Name, LdapName, performs very well when it comes to parsing of Distinguished Names. However, in practical use this implementation has a number of shortcomings:

- The `LdapName` implementation is mutable, which is badly suited for an object representing identity.
- Despite its mutable nature, the API for dynamically building or modifying Distinguished Names using `LdapName` is cumbersome. Extracting values of indexed or (particularly) named components is also a little bit awkward.
- Many of the operations on `LdapName` throw checked Exceptions, requiring try-catch statements for situations where the error is typically fatal and cannot be repaired in a meaningful manner.

To simplify working with Distinguished Names, Spring LDAP provides an `LdapNameBuilder`, as well as a number of utility methods in `LdapUtils` that helps working with `LdapName`.

2.3.1. Examples

Dynamically building an LdapName using LdapNameBuilder

```
package com.example.repo;
import org.springframework.ldap.support.LdapNameBuilder;
import javax.naming.Name;

public class PersonRepoImpl implements PersonRepo {
  public static final String BASE_DN = "dc=example,dc=com";

  protected Name buildDn(Person p) {
    return LdapNameBuilder.newInstance(BASE_DN)
      .add("c", p.getCountry())
      .add("ou", p.getCompany())
      .add("cn", p.getFullname())
      .build();
  }
  ...
```

Assuming that a Person has the following attributes:

Attribute Name	Attribute Value
`country`	Sweden

Attribute Name	Attribute Value
`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
```

Extracting values from a distinguished name using LdapUtils

```
package com.example.repo;
import org.springframework.ldap.support.LdapNameBuilder;
import javax.naming.Name;
public class PersonRepoImpl implements PersonRepo {
...
protected Person buildPerson(Name dn, Attributes attrs) {
  Person person = new Person();
  person.setCountry(LdapUtils.getStringValue(dn, "c"));
  person.setCompany(LdapUtils.getStringValue(dn, "ou"));
  person.setFullname(LdapUtils.getStringValue(dn, "cn"));
  // Populate rest of person object using attributes.

  return person;
}
```

Since Java version <= 1.4 didn't provide any public Distinguished Name implementation at all, Spring LDAP 1.x provided its own implementation, `DistinguishedName`. This implementation suffered from a couple of shortcomings of its own, and has been deprecated in version 2.0. Users are now recommended to use `LdapName` along with the utilities described above instead.

2.4. Binding and Unbinding

2.4.1. Adding Data

Inserting data in Java LDAP is called binding. This is somewhat confusing, because in LDAP terminology *bind* means something completely different. A JNDI bind performs an LDAP Add operation, associating a new entry with a specified distinguished name with a set of attributes. The following example shows how data is added using `LdapTemplate`:

Adding data using Attributes

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
   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;
   }
}
```

Manual Attributes building is — while dull and verbose — sufficient for many purposes. It is however possible to simplify the binding operation further, as described in Simplifying Attribute Access and Manipulation with DirContextAdapter.

2.4.2. Removing Data

Removing data in Java LDAP is called unbinding. A JNDI unbind performs an LDAP Delete operation, removing the entry associated with a distinguished name from the LDAP tree. The following example shows how data is removed using `LdapTemplate`:

Removing data

```
package com.example.repo;

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

2.5. Updating

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

2.5.1. Updating using rebind

A `rebind` is a very crude way to modify data. It's basically an `unbind` followed by a `bind`. The following example demonstrates the use of `rebind`:

Modifying using rebind

```
package com.example.repo;

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

2.5.2. Updating using modifyAttributes

A more sophisticated way of modifying data is to use `modifyAttributes`. This operation takes an array of explicit attribute modifications and performs these on a specific entry:

Modifying using modifyAttributes

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
   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});
   }
}
```

Building `Attributes` and `ModificationItem` arrays is a lot of work, but as you will see in Simplifying Attribute Access and Manipulation with DirContextAdapter Spring LDAP provides more help for simplifying these operations.

3. Simplifying 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 LDAP entries. Spring LDAP makes use of this feature to return `DirContextAdapter` instances in certain search and lookup operations.

`DirContextAdapter` is a very useful tool for working with LDAP attributes, particularly when adding or modifying data.

3.2. Search and Lookup Using ContextMapper

Whenever an entry is found in the LDAP tree, its attributes and Distinguished Name (DN) will be used by Spring LDAP to construct a `DirContextAdapter`. This enables us to use a `ContextMapper` instead of an `AttributesMapper` to transform found values:

Searching using a ContextMapper

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
   ...
   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());
   }
}
```

A shown above, we can retrieve the attribute values directly by name without having to go through the `Attributes` and `Attribute` 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. `DirContextAdapter` does this for you in the `getStringAttributes()` or `getObjectAttributes()` methods:

Getting multi-value attribute values using getStringAttributes()

```
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;
   }
}
```

3.2.1. The AbstractContextMapper

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

Using an AbstractContextMapper

```
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;
    }
}
```

3.3. Adding and Updating Data Using DirContextAdapter

While very useful when extracting attribute values, `DirContextAdapter` is even more powerful for managing the details involved in adding and updating data.

3.3.1. Adding

Below is an example making use of `DirContextAdapter` to implement an improved implementation of the `create` repository method presented in Adding Data.

Binding using DirContextAdapter

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
   ...
   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);
   }
}
```

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 are not specifying the attributes explicitly.

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.

3.3.2. Updating

We previously saw that updating using `modifyAttributes` is the recommended approach, but that this requires us to perform the task of calculating attribute modifications and constructing `ModificationItem` arrays accordingly. `DirContextAdapter` can do all of this for us:

Updating using using DirContextAdapter

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
   ...
   public void update(Person p) {
      Name dn = buildDn(p);
      DirContextOperations context = ldapTemplate.lookupContext(dn);

      context.setAttributeValue("cn", p.getFullname());
      context.setAttributeValue("sn", p.getLastname());
      context.setAttributeValue("description", p.getDescription());

      ldapTemplate.modifyAttributes(context);
   }
}
```

When no mapper is passed to a `ldapTemplate.lookup()`, 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:

Adding and modifying using DirContextAdapter

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
   private LdapTemplate ldapTemplate;

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

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

3.4. DirContextAdapter and Distinguished Names as Attribute Values.

When managing security groups in LDAP it is very common to have attribute values that represent distinguished names. Since distinguished name equality differs from String equality (e.g. whitespace and case differences are ignored in distinguished name equality), calculating attribute modifications using string equality will not work as expected.

For instance, if a `member` attribute has the value `cn=John Doe,ou=People` and we call `ctx.addAttributeValue("member", "CN=John Doe, OU=People")`, the attribute will now be considered to have two values, even though the strings actually represent the same distinguished name.

As of Spring LDAP 2.0, supplying `javax.naming.Name` instances to the attribute modification methods will make `DirContextAdapter` use distinguished name equality when calculating attribute modifications. If we modify the example above to: `ctx.addAttributeValue("member", LdapUtils.newLdapName("CN=John Doe, OU=People"))`, this will **not** render a modification.

Group Membership Modification using DirContextAdapter

```
public class GroupRepo implements BaseLdapNameAware {
    private LdapTemplate ldapTemplate;
    private LdapName baseLdapPath;

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

    public void setBaseLdapPath(LdapName baseLdapPath) {
        this.setBaseLdapPath(baseLdapPath);
    }

    public void addMemberToGroup(String groupName, Person p) {
        Name groupDn = buildGroupDn(groupName);
        Name userDn = buildPersonDn(
            person.getFullname(),
            person.getCompany(),
            person.getCountry());
```

```
        DirContextOperation ctx = ldapTemplate.lookupContext(groupDn);
        ctx.addAttributeValue("member", userDn);

        ldapTemplate.update(ctx);
    }

    public void removeMemberFromGroup(String groupName, Person p) {
        Name groupDn = buildGroupDn(String groupName);
        Name userDn = buildPersonDn(
            person.getFullname(),
            person.getCompany(),
            person.getCountry());

        DirContextOperation ctx = ldapTemplate.lookupContext(groupDn);
        ctx.removeAttributeValue("member", userDn);

        ldapTemplate.update(ctx);
    }

    private Name buildGroupDn(String groupName) {
        return LdapNameBuilder.newInstance("ou=Groups")
            .add("cn", groupName).build();
    }

    private Name buildPersonDn(String fullname, String company, String country) {
        return LdapNameBuilder.newInstance(baseLdapPath)
            .add("c", country)
            .add("ou", company)
            .add("cn", fullname)
            .build();
    }
}
```

In the example above we are implementing `BaseLdapNameAware`, in order to get hold of the base LDAP path as described in Obtaining a reference to the base LDAP path. This is necessary because distinguished names as member attribute values must always be absolute from the directory root.

3.5. A Complete PersonRepository Class

To illustrate the usefulness of Spring LDAP and `DirContextAdapter`, below is a complete Person Repository implementation for LDAP:

```
package com.example.repo;
import java.util.List;

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

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.filter.AndFilter;
import org.springframework.ldap.filter.EqualsFilter;
import org.springframework.ldap.filter.WhitespaceWildcardsFilter;

import static org.springframework.ldap.query.LdapQueryBuilder.query;

public class PersonRepoImpl implements PersonRepo {
   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 ldapTemplate.lookup(dn, getContextMapper());
    }

    public List findByName(String name) {
       LdapQuery query = query()
          .where("objectclass").is("person")
          .and("cn").whitespaceWildcardsLike("name");

       return ldapTemplate.search(query, getContextMapper());
    }

    public List findAll() {
       EqualsFilter filter = new EqualsFilter("objectclass", "person");
       return ldapTemplate.search(LdapUtils.emptyPath(), 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) {
       return LdapNameBuilder.newInstance()
         .add("c", country)
         .add("ou", company)
         .add("cn", fullname)
         .build();
    }

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

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

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. Note that using Object-Directory Mapping (ODM), the the library can automatically handle this for you if you annotate your domain classes appropriately.

4. Object-Directory Mapping (ODM)

4.1. Introduction

Object-relational mapping frameworks like Hibernate and JPA offers developers the ability to use annotations to map relational database tables to Java objects. Spring LDAP project offers a similar ability with respect to LDAP directories through a number of methods: in `LdapOperations`

- `<T> T findByDn(Name dn, Class<T> clazz)`
- `<T> T findOne(LdapQuery query, Class<T> clazz)`
- `<T> List<T> find(LdapQuery query, Class<T> clazz)`
- `<T> List<T> findAll(Class<T> clazz)`
- `<T> List<T> findAll(Name base, SearchControls searchControls, Class<T> clazz)`
- `<T> List<T> findAll(Name base, Filter filter, SearchControls searchControls, Class<T> clazz)`
- `void create(Object entry)`
- `void update(Object entry)`
- `void delete(Object entry)`

4.2. Annotations

Entity classes managed used with the object mapping methods are required to be annotated with annotations from 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.
- `@DnAttribute` - Indicates the mapping of a dn attribute to the object class field.
- `@Transient` - Indicates the field is not persistent and should be ignored by the `OdmManager`.

The `@Entry` and `@Id` annotations are required to be declared on managed classes. `@Entry` is used to specify which object classes the entity maps to and (optionally) the directory root of the LDAP entries represented by the class. All object classes for which fields are mapped are required to be declared. Note that when creating new entries of the managed class, only the declared objectclasses will be used.

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. For example: let's assume that you have entries in your LDAP tree that have the objectclasses `inetOrgPerson,organizationalPerson,person,top`. If you are only interested in changing the attributes defined in the `person` objectclass, your `@Entry` annotation can be `@Entry(objectClasses = { "person", "top" })`. However, if you want to manage attributes defined in the `inetOrgPerson` objectclass you'll need to use the full monty: `@Entry(objectClasses = { "inetOrgPerson", "organizationalPerson", "person", "top" })`.

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`.

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 `@DnAttribute` annotation is used to map object class fields to and from components in the distinguished name of an entry. Fields annotated with `@DnAttribute` will automatically be populated with the appropriate value from the distinguished name when an entry is read

from the directory tree. If the `index` attribute of all `@DnAttribute` annotations in a class is specified, the DN will also be automatically calculated when creating and updating entries. For update scenarios, this will also automatically take care of moving entries in the tree if attributes that are part of the distinguished name have changed.

The `@Transient` annotation is used to indicate the field should be ignored by the object directory mapping and not mapped to an underlying LDAP property. Note that if a `@DnAttribute` is not to be bound to an Attribute, i.e. it is only part of the Distinguished Name and not represented by an object attibute, it must also be annotated with `@Transient`.

4.3. Execution

When all components have been properly configured and annotated, the object mapping methods of `LdapTemplate` can be used as follows:

Execution

```
@Entry(objectClasses = { "person", "top" }, base="ou=someOu")
public class Person {
   @Id
   private Name dn;

   @Attribute(name="cn")
   @DnAttribute(value="cn", index=1)
   private String fullName;

   // No @Attribute annotation means this will be bound to the LDAP attribute
   // with the same value
   private String description;

   @DnAttribute(value="ou", index=0)
   @Transient
   private String company;

   @Transient
   private String someUnmappedField;
   // ...more attributes below
}


public class OdmPersonRepo {
   @Autowired
   private LdapTemplate ldapTemplate;

   public Person create(Person person) {
      ldapTemplate.create(person);
      return person;
   }

   public Person findByUid(String uid) {
      return ldapTemplate.findOne(query().where("uid").is(uid), Person.class);
   }

   public void update(Person person) {
      ldapTemplate.update(person);
   }

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

   public List<Person> findAll() {
      return ldapTemplate.findAll(Person.class);
   }

   public List<Person> findByLastName(String lastName) {
      return ldapTemplate.find(query().where("sn").is(lastName), Person.class);
   }
}
```

4.4. ODM and Distinguished Names as Attribute Values.

Security groups in LDAP commonly contains a multi-value attribute where each of the values is the distinguished name of a user in the system. The difficulties involved when handling these kinds of attributes are discussed in DirContextAdapter and Distinguished Names as Attribute Values..

ODM also has support for `javax.naming.Name` attribute values, making group modifications very easy:

Example Group representation

```
@Entry(objectClasses = {"top", "groupOfUniqueNames"}, base = "cn=groups")
public class Group {

    @Id
    private Name dn;

    @Attribute(name="cn")
    @DnAttribute("cn")
    private String name;

    @Attribute(name="uniqueMember")
    private Set<Name> members;

    public Name getDn() {
        return dn;
    }

    public void setDn(Name dn) {
        this.dn = dn;
    }

    public Set<Name> getMembers() {
        return members;
    }

    public void setMembers(Set<Name> members) {
        this.members = members;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void addMember(Name member) {
        members.add(member);
    }

    public void removeMember(Name member) {
        members.remove(member);
    }
}
```

Modifying group members using `setMembers`, `addMember` and `removeMember` above, and then calling `ldapTemplate.update()`, attribute modifications will be calculated using distinguished name equality, meaning that the text formatting of distinguished names will be disregarded when figuring out whether they are equal.

5. Advanced LDAP Queries

5.1. LDAP Query Builder Parameters

The `LdapQueryBuilder` and its associated classes are intended to support all parameters that can be supplied to an LDAP search. The following parameters are supported:

- `base` - specifies the root DN in the LDAP tree where the search should start.

- `searchScope` - specifies how deep into the LDAP tree the search should traverse.
- `attributes` - specifies the attributes to return from the search. Default is all.
- `countLimit` - specifies the maximum number of entries to return from the search.
- `timeLimit` - specifies the maximum time that the search may take.
- Search filter - the conditions that the entries we are looking for must meet.

An `LdapQueryBuilder` is created with a call to the `query` method of `LdapQueryBuilder`. It's intended as a fluent builder API, where the base parameters are defined first, followed by the filter specification calls. Once filter conditions have been started to be defined with a call to the `where` method of `LdapQueryBuilder`, later attempts to call e.g. `base` will be rejected. The base search parameters are optional, but at least one filter specification call is required.

Search for all entries with objectclass person

```
import static org.springframework.ldap.query.LdapQueryBuilder.query;
...

List<Person> persons = ldapTemplate.search(
      query().where("objectclass").is("person"),
      new PersonAttributesMapper());
```

Search for all entries with objectclass person and cn=John Doe

```
import static org.springframework.ldap.query.LdapQueryBuilder.query;
...

List<Person> persons = ldapTemplate.search(
      query().where("objectclass").is("person")
             .and("cn").is("John Doe"),
      new PersonAttributesMapper());
```

Search for all entries with objectclass person starting at dc=261consulting,dc=com

```
import static org.springframework.ldap.query.LdapQueryBuilder.query;
...

List<Person> persons = ldapTemplate.search(
      query().base("dc=261consulting,dc=com")
             .where("objectclass").is("person"),
      new PersonAttributesMapper());
```

Search for all entries with objectclass person starting at dc=261consulting,dc=com, only returning the cn attribute

```
import static org.springframework.ldap.query.LdapQueryBuilder.query;
...

List<Person> persons = ldapTemplate.search(
      query().base("dc=261consulting,dc=com")
             .attributes("cn")
             .where("objectclass").is("person"),
      new PersonAttributesMapper());
```

Search with or criteria

```
import static org.springframework.ldap.query.LdapQueryBuilder.query;
...
List<Person> persons = ldapTemplate.search(
      query().where("objectclass").is("person"),
             .and(query().where("cn").is("Doe").or("cn").is("Doo"));
      new PersonAttributesMapper());
```

5.2. Filter Criteria

The examples above demonstrates simple equals conditions in LDAP filters. The LDAP query builder has support for the following criteria types:

- `is` - specifies an equals condition (=).
- `gte` - specifies a greater than or equals condition (>=).
- `lte` - specifies a less than or equals condition (< =).
- `like` - specifies a "like" condition where wildcards can be included in the query, e.g. `where("cn").like("J*hn Doe")` will result int the filter `(cn=J*hn Doe)`.
- `whitespaceWildcardsLike` - specifies a condition where all whitespace is replaced with wildcards, e.g. `where("cn").whitespaceWildcardsLike("John Doe")` will result in the filter `(cn=*John*Doe*)`.
- `isPresent` - specifies condition that checks for the presence of an attribute, e.g. `where("cn").isPresent()` will result in the filter `(cn=*)`.
- `not` - specifies that the current condition should be negated, e.g. `where("sn").not().is("Doe)` will result in the filter `(!(sn=Doe))`

5.3. Hardcoded Filters

There are occasions when you will want to specify a hardcoded filter as input to an `LdapQuery`. `LdapQueryBuilder` has two methods for this purpose:

- `filter(String hardcodedFilter)` - uses the specified string as filter. Note that the specified input string will not be touched in any way, meaning that this method is not particularly well suited if you are building filters from user input.
- `filter(String filterFormat, String... params)` - uses the specified string as input to `MessageFormat`, properly encoding the parameters and inserting them at the specified places in the filter string.

You cannot mix the hardcoded filter methods with the `where` approach described above; it's either one or the other. What this means is that if you specified a filter using `filter()` you will get an exception if you try to call `where` afterwards.

---

6. Configuration

6.1. Introduction

The recommended way of configuring Spring LDAP is using the custom XML configuration namespace. In order to make this available you need to include the Spring LDAP namespace declaration in your bean file, e.g.:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ldap="http://www.springframework.org/schema/ldap"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/ldap http://www.springframework.org/schema/ldap/spring-ldap.xsd">
```

6.2. ContextSource Configuration

The `ContextSource` is defined using a `<ldap:context-source>` tag. The simplest possible `context-source` declaration requires you to specify a server url, a username, and a password:

Simplest possible context-source declaration

```
<ldap:context-source
    username="cn=Administrator"
    password="secret"
    url="ldap://localhost:389" />
```

This will create an `LdapContextSource` with default values (see below), and the url and authentication credentials as specified. The configurable attributes on context-source are as follows (required attributes marked with *):

Table 1. ContextSource Configuration Attributes

Attribute	Default	Description
`id`	`contextSource`	The id of the created bean.
`username`		The username (principal) to use when authenticating with the LDAP server. This will usually be the distinguished name of an admin user (e.g. `cn=Administrator`), but may differ depending on server and authentication method. Required if `authentication-source-ref` is not explicitly configured.
`password`		The password (credentials) to use when authenticating with the LDAP server. Required if `authentication-source-ref` is not explicitly configured.
`url` *		The URL of the LDAP server to use. 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`. If fail-over functionality is desired, more than one URL can be specified, separated using comma (,).
`base`	`LdapUtils.emptyLdapName()`	The base DN. When this attribute has been configured, all Distinguished Names supplied to and received from LDAP operations will be relative to the specified LDAP path. 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 Obtaining a reference to the base LDAP path
`anonymous-read-only`	`false`	Defines whether read-only operations will be performed using an anonymous (unauthenticated) context. **Note** that setting this parameter to `true` together with the compensating transaction support is not supported and will be rejected.
`referral`	`null`	Defines the strategy to handle referrals, as described here. Valid values are: • `ignore` • `follow` • `throw`

Attribute	Default	Description
`native-pooling`	`false`	Specify whether native Java LDAP connection pooling should be used. Consider using Spring LDAP connection pooling instead. See Pooling Support for more information.
`authentication-source-ref`	A `SimpleAuthenticationSource` instance.	Id of the AuthenticationSource instance to use (see below).
`authentication-strategy-ref`	A `SimpleDirContextAuthenticationStrategy` instance.	Id of the DirContextAuthenticationStrategy instance to use (see below).
`base-env-props-ref`		Reference to a Map of custom environment properties that should supplied with the environment sent to the `DirContext` on construction.

6.2.1. 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.

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 User Authentication using Spring LDAP.

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

If `username` is the dn of an LDAP user, it 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 specified on the `context-source` element.

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

Custom DirContext Authentication Processing

The default authentication mechanism used in Spring LDAP is `SIMPLE` authentication. This means that the principal (as specified to the `username` attribute) 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 reference to the `context-source` element using the `authentication-strategy-ref` attribute.

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 `usernmae` 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.

When working with TLS connections you need to make sure that the native LDAP Pooling functionality (as specified using the `native-pooling` attribute is turned off. 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 Pooling Support.

Custom Principal and Credentials ManagementUsing the

While the user name (i.e. user DN) and password used for creating an authenticated `Context` are statically defined by default - the ones defined in the `context-source` element configuration will be used throughout the lifetime of the `ContextSource` - there are several cases where 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 reference to an `AuthenticationSource` implementation to the `context-source` element using the `authentication-source-ref` element, instead of explicitly specifying the `username` 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.

Using the SpringSecurityAuthenticationSource

```
<beans>
...
    <ldap:context-source
        url="ldap://localhost:389"
        authentication-source-ref="springSecurityAuthenticationSource"/>

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

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

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

6.2.2. 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 off. 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.

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 Pooling Support. If pooling functionality is required, this is the recommended approach.

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.

6.2.3. Advanced ContextSource Configuration

Custom DirContext Environment Properties

In some cases the user might want to specify additional environment setup properties in addition to the ones directly configurable on `context-source`. Such properties should be set in a `Map` and referenced in the `base-env-props-ref` attribute.

6.3. LdapTemplate Configuration

The `LdapTemplate` is defined using a `<ldap:ldap-template>` tag. The simplest possible `ldap-template` declaration is the simple tag:

Simplest possible ldap-template declaration

```
<ldap:ldap-template />
```

This will create an `LdapTemplate` instance with the default id, referencing the default `ContextSource`, which is expected to have the id `contextSource` (the default for the `context-source` element).

The configurable attributes on `ldap-template` are as follows:

Table 2. LdapTemplate Configuration Attributes

Attribute	Default	Description
`id`	`ldapTemplate`	The id of the created bean.
`context-source-ref`	`contextSource`	Id of the ContextSource instance to use.
`count-limit`	`0`	The default count limit for searches. 0 means no limit.
`time-limit`	`0`	The default time limit for searches in milliseconds. 0 means no limit.
`search-scope`	`SUBTREE`	The default search scope for searches. Valid values are: • `OBJECT` • `ONELEVEL` • `SUBTREE`
`ignore-name-not-found`	`false`	Specifies whether NameNotFoundException should be ignored in searches. Setting this attribute to true will cause errors caused by invalid search base to be silently swallowed.
`ignore-partial-result`	`false`	Specifies whether PartialResultException should be ignored in searches. Some LDAP servers have problems with referrals; these should normally be followed automatically, but if this doesn't work it will manifest itself with a PartialResultException. Setting this attribute to true presents a work-around to this problem.
`odm-ref`		Id of the ObjectDirectoryMapper instance to use. Default is a default-configured DefaultObjectDirectoryMapper.

6.4. 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 `BaseLdapNameAware` interface. Secondly, a `BaseLdapPathBeanPostProcessor` needs to be defined in the application context:

Implementing BaseLdapNameAware

```
package com.example.service;
public class PersonService implements PersonService, BaseLdapNameAware {
   ...
   private LdapName basePath;

   public void setBaseLdapPath(LdapName basePath) {
      this.basePath = basePath;
   }
   ...
   private LdapName getFullPersonDn(Person person) {
      return LdapNameBuilder.newInstance(basePath)
          .add(person.getDn())
          .build();
   }
   ...
}
```

Specifying a BaseLdapPathBeanPostProcessor in your ApplicationContext

```
<beans>
   ...
   <ldap:context-source
          username="cn=Administrator"
          password="secret"
          url="ldap://localhost:389"
          base="dc=261consulting,dc=com" />
   ...
   <bean class="org.springframework.ldap.core.support.BaseLdapPathBeanPostProcessor" />
</beans>
```

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.

7. Spring LDAP Repositories

7.1. Overview

Spring LDAP has built-in support for Spring Data repositories. The basic functionality and configuration is described here. When working with Spring LDAP repositories, please note the following:

- Spring LDAP repositories can be enabled using an `<ldap:repositories>` tag in your XML configuration or using an `@EnableLdapRepositories` annotation on a configuration class.
- To include support for `LdapQuery` parameters in automatically generated repositories, have your interface extend `LdapRepository` rather than `CrudRepository`.
- All Spring LDAP repositories must work with entities annotated with the ODM annotations, as described in Object-Directory Mapping (ODM).
- Since all ODM managed classes must have a Distinguished Name as ID, all Spring LDAP repositories must have the ID type parameter set to `javax.naming.Name`. Indeed, the built-in `LdapRepository` only takes one type parameter; the managed entity class, defaulting ID to `javax.naming.Name`.
- Due to specifics of the LDAP protocol, paging and sorting is not supported for Spring LDAP repositories.

7.2. QueryDSL support

Basic QueryDSL support is included in Spring LDAP. This support includes the following:

- An Annotation Processor, `LdapAnnotationProcessor`, for generating QueryDSL classes based on Spring LDAP ODM annotations. See Object-Directory Mapping (ODM) for more information on the ODM annotations.
- A Query implementation, `QueryDslLdapQuery`, for building and executing QueryDSL queries in code.
- Spring Data repository support for QueryDSL predicates. `QueryDslPredicateExecutor` includes a number of additional methods with appropriate parameters; extend this interface along with `LdapRepository` to include this support in your repository.

---

8. Pooling Support

8.1. 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 supplying a `<ldap:pooling />` sub-element to the `<ldap:context-source />` element in the application context configuration. Read-only and read-write `DirContext` objects are pooled separately (if `anonymous-read-only` is specified. Jakarta Commons-Pool is used to provide the underlying pool implementation.

8.2. 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.

If connection validation is configured, pooled connections are validated using `DefaultDirContextValidator`. `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 default settings should work with no configuration changes on most LDAP servers and provide the fastest way to validate the `DirContext`. If customization is required this can be done using the validation configuration attributes, described below.

Connections will be automatically invalidated if they throw an exception that is considered non-transient. E.g. if a `DirContext` instance throws a `javax.naming.CommunicationException`, this will be interpreted as a non-transient error and the instance will be automatically invalidated, without the overhead of an additional testOnReturn operation. The exceptions that are interpreted as non-transient are configured using the `nonTransientExceptions` property of the `PoolingContextSource`.

8.3. Pool Configuration

The following attributes are available on the `<ldap:pooling />` element for configuration of the DirContext pool:

Table 3. Pooling Configuration Attributes

Attribute	Default	Description
`max-active`	`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.

Attribute	Default	Description
`max-total`	`-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.
`max-idle`	`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.
`min-idle`	`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.
`max-wait`	`-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.
`when-exhausted`	`BLOCK`	Specifies the behaviour when the pool is exhausted. • The `FAIL` option will throw a `NoSuchElementException` when the pool is exhausted. • The `BLOCK` option will wait until a new object is available. If `max-wait` is positive a `NoSuchElementException` is thrown if no new object is available after the `max-wait` time expires. • The `GROW` option will create and return a new object (essentially making `max-active` meaningless).
`test-on-borrow`	`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.
`test-on-return`	`false`	The indication of whether objects will be validated before being returned to the pool.
`test-while-idle`	`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.

Attribute	Default	Description
`eviction-run-interval-millis`	`-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.
`tests-per-eviction-run`	`3`	The number of objects to examine during each run of the idle object evictor thread (if any).
`min-evictable-time-millis`	`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).
`validation-query-base`	`LdapUtils.emptyName()`	The search base to be used when validating connections. Only used if `test-on-borrow`, `test-on-return`, or `test-while-idle` is specified
`validation-query-filter`	`objectclass=*`	The search filter to be used when validating connections. Only used if `test-on-borrow`, `test-on-return`, or `test-while-idle` is specified
`validation-query-search-controls-ref`	`null`; default search control settings are described above.	Id of a SearchControls instance to be used when validating connections. Only used if `test-on-borrow`, `test-on-return`, or `test-while-idle` is specified
`non-transient-exceptions`	`javax.naming.CommunicationException`	Comma-separated list of Exception classes. The listed exceptions will be considered non-transient with regards to eager invalidation. Should any of the listed exceptions (or subclasses of them) be thrown by a call to a pooled `DirContext` instance, that object will be automatically invalidated without any additional testOnReturn operation.

8.4. Configuration

Configuring pooling requires adding an `<ldap:pooling>` element nested in the `<ldap:context-source>` element:

```
<beans>
   ...
    <ldap:context-source
        password="secret" url="ldap://localhost:389" username="cn=Manager">
        <ldap:pooling />
    </ldap:context-source>
   ...
</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.

8.4.1. Validation Configuration

```
<beans>
   ...
    <ldap:context-source
        username="cn=Manager" password="secret" url="ldap://localhost:389" >
        <ldap:pooling
            test-on-borrow="true"
            test-while-idle="true" />
    </ldap:context-source>
   ...
</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.

8.5. Known Issues

8.5.1. 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.

9. Adding Missing Overloaded API Methods

9.1. 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.

A custom search method using SearchExecutor and AttributesMapper

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
   ...
   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();
    }
}
```

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

A custom search method using SearchExecutor and ContextMapper

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
    ...
    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();
    }
}
```

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

9.2. 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:

A custom DirContext method using ContextExecutor

```
package com.example.repo;

public class PersonRepoImpl implements PersonRepo {
    ...
    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.

10. Processing the DirContext

10.1. 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)
```

10.2. 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:

A request control DirContextProcessor implementation

```
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());
            }
        }
    }
}
```

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`.

10.3. 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 using the class `PagedResultsDirContextProcessor`. The `PagedResultsDirContextProcessor` class creates a `PagedResultsControl` with the requested page size and adds it to the `LdapContext`. After the search, it gets the `PagedResultsResponseControl` and retrieves the paged results cookie, which is needed to keep the context between consecutive paged results requests.

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

Paged results using PagedResultsDirContextProcessor

```
public List<String> getAllPersonNames() {
```

```
final SearchControls searchControls = new SearchControls();
searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);

final PagedResultsDirContextProcessor processor =
      new PagedResultsDirContextProcessor(PAGE_SIZE);

return SingleContextSource.doWithSingleContext(
      contextSource, new LdapOperationsCallback<List<String>>() {

    @Override
    public List<String> doWithLdapOperations(LdapOperations operations) {
      List<String> result = new LinkedList<String>();

      do {
        List<String> oneResult = operations.search(
          "ou=People",
          "(&(objectclass=person))",
          searchControls,
          CN_ATTRIBUTES_MAPPER,
          processor);
        result.addAll(oneResult);
      } while(processor.hasMore());

      return result;
    }
  });
}
```

In order for a paged results cookie to continue being valid, it is imperative that the same underlying connection is used for each paged results call. This can be accomplished using the `SingleContextSource`, as demonstrated in the example.

11. Transaction Support

11.1. 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 no LDAP 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.

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.

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) very little will be gained by using the LDAP transaction support.

11.2. Configuration

Configuring Spring LDAP transactions should look very familiar if you're used to configuring Spring transactions. You will annotate your transacted classes with `@Transactional`, create a `TransactionManager` instance and include a `<tx:annotation-driven>` tag in your bean configuraion.

```
<ldap:context-source
       url="ldap://localhost:389"
       base="dc=example,dc=com"
       username="cn=Manager"
       password="secret" />

<ldap:ldap-template id="ldapTemplate" />
<ldap:transaction-manager>
    <!--
    Note this default configuration will not work for more complex scenarios;
    see below for more information on RenamingStrategies.
    -->
   <ldap:default-renaming-strategy />
</ldap:transaction-manager>

<!--
   The MyDataAccessObject class is annotated with @Transactional.
-->
<bean id="myDataAccessObject" class="com.example.MyRepository">
  <property name="ldapTemplate" ref="ldapTemplate" />
</bean>

<tx:annotation-driven />
...
```

While this 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 Renaming Strategies below.

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

11.3. 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 by supplying a `data-source-ref` attribute to the `<ldap:transaction-manager>` tag. This will create a `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:

```
<ldap:transaction-manager data-source-ref="dataSource" >
  <ldap:default-renaming-strategy />
<ldap:transaction-manager />
```

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.

The same thing can be accomplished for Hibernate integration by supplying a `session-factory-ref` attribute to the `<ldap:transaction-manager>` tag.

```
<ldap:transaction-manager session-factory-ref="dataSource" >
  <ldap:default-renaming-strategy />
<ldap:transaction-manager />
```

11.4. 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:

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`, and calculate a temporary DN.	Rename the entry to a temporary location.	Bind the new `Attributes` at the original DN, and unbind the original entry from its temporary location.	Rename the entry from the temporary location back to its original DN.
`modifyAttributes`	Make record of the DN of the entry to modify and calculate compensating `ModificationItem`s for the modifications to be done.	Perform the `modifyAttributes` operation.	No operation.	Perform a `modifyAttributes` operation using the calculated compensating `ModificationItem`s.

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

11.4.1. 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` specified in a sub-element to the `<ldap:transaction-manager >` declaration in the configuration. Two implementations are supplied with Spring LDAP:

- `DefaultTempEntryRenamingStrategy` (the default). Specified using a `<ldap:default-renaming-strategy />` element. 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 `temp-suffix` attribute.

- `DifferentSubtreeTempEntryRenamingStrategy`. Specified using a `<ldap:different-subtree-renaming-strategy />` element. 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 `subtree-node` attribute. E.g., if `subtree-node` 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.

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`.

12. User Authentication using Spring LDAP

12.1. 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) {
  List<String> result = ldapTemplate.search(
      query().where("uid").is(uid),
      new AbstractContextMapper() {
         protected String doMapFromContext(DirContextOperations ctx) {
            return ctx.getNameInNamespace();
         }
      });

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

  return 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 includes an authenticate method in LdapTemplate that provide this functionality: `boolean authenticate(LdapQuery query, String password);`

Using this method authentication becomes as simple as this:

Authenticating a user using Spring LDAP.

```
ldapTemplate.authenticate(query().where("uid").is("john.doe"), "secret");
```

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

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

12.2. 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 includes the callback interface `AuthenticatedLdapEntryContextMapper` and a corresponding `authenticate` method: `<T> T authenticate(LdapQuery query, String password, AuthenticatedLdapEntryContextMapper<T> mapper);`

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

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

```
AuthenticatedLdapEntryContextMapper<DirContextOperations> mapper = new AuthenticatedLdapEntryContextMapper<DirContextOperations>() {
  public DirContextOperations mapWithContext(DirContext ctx, LdapEntryIdentification ldapEntryIdentification) {
    try {
      return (DirContextOperations) ctx.lookup(ldapEntryIdentification.getRelativeName());
    }
    catch (NamingException e) {
      throw new RuntimeException("Failed to lookup " + ldapEntryIdentification.getRelativeName(), e);
    }
  }
};

ldapTemplate.authenticate(query().where("uid").is("john.doe"), "secret", mapper);
```

12.3. Obsolete authentication methods

In addition to the `authenticate` methods described above there are a number of deprecated methods that can be used for authentication.

While these will work fine, the recommendation is to use the `LdapQuery` methods instead.

12.4. 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.

---

13. LDIF Parsing

13.1. 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.

13.2. 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 `javax.naming.ldap.LdapName` class.

13.3. 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).

13.4. 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.

13.5. 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.

---

14. Utilities

14.1. 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:

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

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