File: tutorial — ActiveLdap ActiveLdap

ActiveLdap ActiveLdap Fabrication

Index » File: tutorial
(no frames)
Class List Method List File List

Table of Contents (left)

About Project

- Project page
- Mailinglist
- GitHub page
- GoogleCode page

About ActiveLdap

- Reference
- Tutorial
- Rails
- Install
- Development

About ActiveLdap Fabrication

- Reference

Tutorial

Introduction

ActiveLdap is a novel way of interacting with LDAP. Most interaction with LDAP is done using clunky LDIFs, web interfaces, or with painful APIs that required a thick reference manual nearby. ActiveLdap aims to fix that. Inspired by ActiveRecord, ActiveLdap provides an object oriented interface to LDAP entries.

The target audience is system administrators and LDAP users everywhere that need quick, clean access to LDAP in Ruby.

What’s LDAP?

LDAP stands for “Lightweight Directory Access Protocol.” Basically this means that it is the protocol used for accessing LDAP servers. LDAP servers lightweight directories. An LDAP server can contain anything from a simple digital phonebook to user accounts for computer systems. More and more frequently, it is being used for the latter. My examples in this text will assume some familiarity with using LDAP as a centralized authentication and authorization server for Unix systems. (Unfortunately, I've yet to try this against Microsoft's ActiveDirectory, despite what the name implies.)

Further reading:

- RFC1777 – Lightweight Directory Access Protocol
- OpenLDAP

So why use ActiveLdap?

Using LDAP directly (even with the excellent Ruby/LDAP), leaves you bound to the world of the predefined LDAP API. While this API is important for many reasons, having to extract code out of LDAP search blocks and create huge arrays of LDAP.mod entries make code harder to read, less intuitive, and just less fun to write. Hopefully, ActiveLdap will remedy all of these problems!

Getting Started

Requirements

- A Ruby implementation: Ruby 1.8.x, 1.9.1 or JRuby
- A LDAP library: Ruby/LDAP (for Ruby), Net::LDAP (for Ruby or JRuby) or JNDI (for JRuby)
- A LDAP server: OpenLDAP, etc
  - Your LDAP server must allow root_dse queries to allow for schema queries

Installation

Assuming all the requirements are installed, you can install by gem.

```
# gem install activeldap
```

Now as a quick test, you can run:

```
$ irb -rubygems
irb> require 'active_ldap'
=> true
irb> exit
```

If the require returns false or an exception is raised, there has been a problem with the installation. You may need to customize what setup.rb does on install.

Usage

This section covers using ActiveLdap from writing extension classes to writing applications that use them.

Just to give a taste of what's to come, here is a quick example using irb:

```
irb> require 'active_ldap'
```

Call setup_connection method for connect to LDAP server. In this case, LDAP server is localhost, and base of LDAP tree is "dc=dataspill,dc=org".

```
irb> ActiveLdap::Base.setup_connection :host => 'local
```

Here's an extension class that maps to the LDAP Group objects:

```
irb> class Group < ActiveLdap::Base
irb*   ldap_mapping
irb* end
```

In the above code, Group class handles sub tree of ou=Groups tha is :base value specified by setup_connection. A instance of Group class represents a LDAP object under ou=Gruops.

Here is the Group class in use:

```
# Get all group names
irb> all_groups = Group.find(:all, '*').collect {|grou
=> ["root", "daemon", "bin", "sys", "adm", "tty", ...

# Get LDAP objects in develop group
irb> group = Group.find("develop")
=> #<Group objectClass:<...> ...>

# Get cn of the develop group
irb> group.cn
=> "develop"

# Get gid_number of the develop group
irb> group.gid_number
=> "1003"
```

That’s it! No let’s get back in to it.

Extension Classes

Extension classes are classes that are subclassed from ActiveLdap::Base. They are used to represent objects in your LDAP server abstractly.

Why do I need them?

Extension classes are what make ActiveLdap “active”! They do all the background work to make easy-to-use objects by mapping the LDAP object’s attributes on to a Ruby class.

Special Methods

I will briefly talk about each of the methods you can use when defining an extension class. In the above example, I only made one special method call inside the Group class. More than likely, you will want to more than that.

ldap_mapping

ldap_mapping is the only required method to setup an extension class for use with ActiveLdap. It must be called inside of a subclass as shown above.

Below is a much more realistic Group class:

```
class Group < ActiveLdap::Base
  ldap_mapping :dn_attribute => 'cn',
               :prefix => 'ou=Groups', :classes => ['
               :scope => :one
end
```

As you can see, this method is used for defining how this class maps in to LDAP. Let’s say that my LDAP tree looks something like this:

```
* dc=dataspill,dc=org
|- ou=People,dc=dataspill,dc=org
|+ ou=Groups,dc=dataspill,dc=org
  \
   |- cn=develop,ou=Groups,dc=dataspill,dc=org
   |- cn=root,ou=Groups,dc=dataspill,dc=org
   |- ...
```

Under ou=People I store user objects, and under ou=Groups, I store group objects. What |ldap_mapping| has done is mapped the class in to the LDAP tree abstractly. With the given :dn_attributes and :prefix, it will only work for entries under ou=Groups,dc=dataspill,dc=org using the primary attribute ‘cn’ as the beginning of the distinguished name.

Just for clarity, here’s how the arguments map out:

```
 cn=develop,ou=Groups,dc=dataspill,dc=org
 ^^         ^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^
:dn_attribute |         |
            :prefix     |
              :base from setup_connection
```

:scope tells ActiveLdap to only search under ou=Groups, and not to look deeper for dn_attribute matches. (e.g. cn=develop,ou=DevGroups,ou=Groups,dc=dataspill,dc=org) You can choose value from between :sub, :one and :base.

Something’s missing: :classes. :classes is used to tell ActiveLdap what the minimum requirement is when creating a new object. LDAP uses objectClasses to define what attributes a LDAP object may have. ActiveLdap needs to know what classes are required when creating a new object. Of course, you can leave that field out to default to [‘top’] only. Then you can let each application choose what objectClasses their objects should have by calling the method e.g. Group#add_class(*values).

Note that is can be very important to define the default :classes value. Due to implementation choices with most LDAP servers, once an object is created, its structural objectclasses may not be removed (or replaced). Setting a sane default may help avoid programmer error later.

:classes isn't the only optional argument. If :dn_attribute is left off, it defaults to super class's value or 'cn'. If :prefix is left off, it will default to 'ou=PluralizedClassName'. In this case, it would be 'ou=Groups'.

:classes should be an Array. :dn_attribute should be a String and so should :prefix.

belongs_to

This method allows an extension class to make use of other extension classes tying objects together across the LDAP tree. Often, user objects will be members of, or belong_to, Group objects.

```
* dc=dataspill,dc=org
|+ ou=People,dc=dataspill,dc=org
 \
  |- uid=drewry,ou=People,dc=dataspill,dc=org
|- ou=Groups,dc=dataspill,dc=org
```

In the above tree, one such example would be user 'drewry' who is a part of the group 'develop'. You can see this by looking at the 'memberUid' field of 'develop'.

```
irb> develop = Group.find('develop')
=> ...
irb> develop.memberUid
=> ['drewry', 'builder']
```

If we look at the LDAP entry for 'drewry', we do not see any references to group 'develop'. In order to remedy that, we can use belongs_to

```
irb> class User < ActiveLdap::Base
irb*   ldap_mapping :dn_attribute => 'uid', :prefix =>
irb*   belongs_to :groups, :class_name => 'Group', :ma
irb* end
```

Now, class User will have a method called ‘groups’ which will retrieve all Group objects that a user is in.

```
irb> me = User.find('drewry')
irb> me.groups
=>  #<ActiveLdap::Association::BelongsToMany...>    #
irb> me.groups.each { |group| p group.cn };nil
"cdrom"
"audio"
"develop"
=> nil
(Note: nil is just there to make the output cleaner..
```

TIP: If you weren’t sure what the distinguished name attribute was for Group, you could also do the following:

```
irb> me.groups.each { |group| p group.id };nil
"cdrom"
"audio"
"develop"
=> nil
```

Now let’s talk about the arguments of belongs_to. We use the following code that extends Group group a bit for explain:

```
class User < ActiveLdap::Base
  ldap_mapping :dn_attribute => 'uid', :prefix => 'Pe

  # Associate with primary belonged group
  belongs_to :primary_group, :foreign_key => 'gidNumbe
             :class_name => 'Group', :primary_key =>

  # Associate with all belonged groups
  belongs_to :groups,  :foreign_key => 'uid',
             :class_name => 'Group', :many => 'membe
end
```

The first argument is the name of the method you wish to create. In this case, we created a method called primary_group and groups using the symbol :primary_group and :groups. The next collection of arguments are actually a Hash (as with ldap_mapping).

:foreign_key tells belongs_to what attribute Group objects have that match the related object's attribute. If :foreign_key is left off of the argument list, it is assumed to be the dn_attribute.

In the example, uid is used for :foreign_key. It may confuse you.

ActiveLdap uses :foreign_key as "own attribute name". So it may not be "foreign key". You can consider :foreign_key just as a relation key.

:primary_key is treated as "related object's attribute name" as we discussed later.

:class_name should be a string that has the name of a class you've already included. If your class is inside of a module, be sure to put the whole name, e.g. `:class_name => "MyLdapModule::Group"`.

:many and :primary_key are similar. Both of them specifies attribute name of related object specified by :foreign_key. Those values are attribute name that can be used by object of class specified by :class_name.

Relation is resolved by searching entries of :class_name class with :foreign_key attribute value. Search target attribute for it is :primary_key or :many. primary_group method in the above example searches Group objects with User object's gidNumber value as Group object's gidNumber value. Matched Group objects are belonged objects.

:parimary_key is used for an object just belongs to an object. The first matched object is treated as beloned object.

:many is used for an object belongs to many objects. All of matched objects are treated as belonged objects.

In addition, you can do simple membership tests by doing the following:

```
irb> me.groups.member? 'root'
=> false
irb> me.groups.member? 'develop'
=> true
```

has_many

has_many

This method is the opposite of belongs_to. Instead of checking other objects in other parts of the LDAP tree to see if you belong to them, you have multiple objects from other trees listed in your object. To show this, we can just invert the example from above:

```
class Group < ActiveLdap::Base
  ldap_mapping :dn_attribute => 'cn', :prefix => 'ou=(

  # Associate with primary belonged users
  has_many :primary_members, :foreign_key => 'gidNumbe
           :class_name => "User", :primary_key => 'gi(

  # Associate with all belonged users
  has_many :members,  :wrap => "memberUid",
           :class_name => "User",  :primary_key => 'u:
end
```

Now we can see that group develop has user ‘drewry’ as a member, and it can even return all responses in object form just like belongs_to methods.

```
irb> develop = Group.find('develop')
=> ...
irb> develop.members
=> #<ActiveLdap::Association::HasManyWrap:..> # Enume:
irb> develop.members.map{|member| member.id}
=> ["drewry", "builder"]
```

The arguments for has_many follow the exact same idea that belongs_to’s arguments followed. :wrap’s contents are used to search for matching :primary_key content. If :primary_key is not specified, it defaults to the dn_attribute of the specified :class_name.

Using these new classes

These new classes have many method calls. Many of them are automatically generated to provide access to the LDAP object’s attributes. Other were defined during class creation by special methods like belongs_to. There are a few other methods that do not fall in to these categories.

.find

.find is a class method that is accessible from any subclass of Base that has ‘ldap_mapping’ called. When called .first(:first) returns that has 'ldap_mapping' called. When called .first(:first) returns the first match of the given class.

```
irb> Group.find(:first, 'deve*").cn
=> "develop"
```

In this simple example, Group.find took the search string of ‘deve*’ and searched for the first match in Group where the dn_attribute matched the query. This is the simplest example of .find.

```
irb> Group.find(:all).collect {|group| group.cn}
=> ["root", "daemon", "bin", "sys", "adm", "tty", ...
```

Here .find(:all) returns all matches to the same query. Both .find(:first) and .find(:all) also can take more expressive arguments:

```
irb> Group.find(:all, :attribute => 'gidNumber', :valu
=> ["develop"]
```

So it is pretty clear what :attribute and :value do – they are used to query as :attribute=:value.

If :attribute is unspecified, it defaults to the dn_attribute.

It is also possible to override :attribute and :value by specifying :filter. This argument allows the direct specification of a LDAP filter to retrieve objects by.

.search

.search is a class method that is accessible from any subclass of Base, and Base. It lets the user perform an arbitrary search against the current LDAP connection irrespetive of LDAP mapping data. This is meant to be useful as a utility method to cover 80% of the cases where a user would want to use Base.connection directly.

```
irb> Base.search(:base => 'dc=example,dc=com', :filte
                 :scope => :sub, :attributes => ['uid
=>  [["uid=root,ou=People,dc=dataspill,dc=org",{"cn"=
```

You can specify the :filter, :base, :scope, and :attributes, but they all have defaults —

- :filter defaults to objectClass=* – usually this isn't what you want
- :base defaults to the base of the class this is executed from (as set in ldap_mapping)
- :scope defaults to :sub. Usually you won't need to change it (You can choose value also from between :one and :base)
- :attributes defaults to [] and is the list of attributes you want back. Empty means all of them.

#valid?

valid? is a method that verifies that all attributes that are required by the objects current objectClasses are populated.

#save

save is a method that writes any changes to an object back to the LDAP server. It automatically handles the addition of new objects, and the modification of existing ones.

.exists?

exists? is a simple method which returns true is the current object exists in LDAP, or false if it does not.

```
irb> User.exists?("dshadsadsa")
=> false
```

ActiveLdap::Base

ActiveLdap::Base has come up a number of times in the examples above. Every time, it was being used as the super class for the wrapper objects. While this is it's main purpose, it also handles quite a bit more in the background.

What is it?

ActiveLdap::Base is the heart of ActiveLdap. It does all the schema parsing for validation and attribute-to-method mangling as well as manage the connection to LDAP.

setup_connection

Base.setup_connection takes many (optional) arguments and is

Base.setup_connection takes many (optional) arguments and is used to connect to the LDAP server. Sometimes you will want to connect anonymously and other times over TLS with user credentials. Base.setup_connection is here to do all of that for you.

By default, if you call any subclass of Base, such as Group, it will call Base.setup_connection() if these is no active LDAP connection. If your server allows anonymous binding, and you only want to access data in a read-only fashion, you won’t need to call Base.setup_connection. Here is a fully parameterized call:

```
Base.setup_connection(
  :host => 'ldap.dataspill.org',
  :port => 389,
  :base => 'dc=dataspill,dc=org',
  :logger => logger_object,
  :bind_dn => "uid=drewry,ou=People,dc=dataspill,dc=o
  :password_block => Proc.new { 'password12345' },
  :allow_anonymous => false,
  :try_sasl => false
)
```

There are quite a few arguments, but luckily many of them have safe defaults:

- :host defaults to “127.0.0.1”.
- :port defaults to nil. 389 is applied if not specified.
- :bind_dn defaults to nil. anonymous binding is applied if not specified.
- :logger defaults to a Logger object that prints fatal messages to stderr
- :password_block defaults to nil
- :allow_anonymous defaults to true
- :try_sasl defaults to false – see Advanced Topics for more on this one.

Most of these are obvious, but I’ll step through them for completeness:

- :host defines the LDAP server hostname to connect to.
- :port defines the LDAP server port to connect to.
- :method defines the type of connection – :tls, :ssl, :plain
- :base specifies the LDAP search base to use with the prefixes defined in all subclasses.
- :bind_dn specifies what your server expects when attempting to bind with credentials.
- :logger accepts a custom logger object to integrate with any other logging your application uses.
- :password_block, if defined, give the Proc block for

acquiring the password
- :password, if defined, give the user's password as a String
- :store_password indicates whether the password should be stored, or if used whether the :password_block should be called on each reconnect.
- :allow_anonymous determines whether anonymous binding is allowed if other bind methods fail
- :try_sasl, when true, tells ActiveLdap to attempt a SASL-GSSAPI bind
- :sasl_quiet, when true, tells the SASL libraries to not spew messages to STDOUT
- :sasl_options, if defined, should be a hash of options to pass through. This currently only works with the ruby-ldap adapter, which currently only supports :realm, :authcid, and :authzid.
- :retry_limit – indicates the number of attempts to reconnect that will be undertaken when a stale connection occurs. -1 means infinite.
- :retry_wait – seconds to wait before retrying a connection
- :scope – dictates how to find objects. (Default: :one)
- :timeout – time in seconds – defaults to disabled. This CAN interrupt search() requests. Be warned.
- :retry_on_timeout – whether to reconnect when timeouts occur. Defaults to true See lib/configuration.rb(ActiveLdap::Configuration::DEFAULT_ for defaults for each option

Base.setup_connection just setups connection configuration. A connection is connected and bound when it is needed. It follows roughly the following approach:

- Connect to host:port using :method

- If bind_dn and password_block/password, attempt to bind with credentials.
- If that fails or no password_block and anonymous allowed, attempt to bind anonymously.
- If that fails, error out.

On connect, the configuration options passed in are stored in an internal class variable which is used to cache the information without ditching the defaults passed in from configuration.rb

connection

Base.connection returns the ActiveLdap::Connection object.

Exceptions

There are a few custom exceptions used in ActiveLdap. They are detailed below.

DeleteError

This exception is raised when #delete fails. It will include LDAP error information that was passed up during the error.

SaveError

This exception is raised when there is a problem in #save updating or creating an LDAP entry. Often the error messages are cryptic. Looking at the server logs or doing an Wireshark dump of the connection will often provide better insight.

AuthenticationError

This exception is raised during Base.setup_connection if no valid authentication methods succeeded.

ConnectionError

This exception is raised during Base.setup_connection if no valid connection to the LDAP server could be created. Check you Base.setup_connection arguments, and network connectivity! Also check your LDAP server logs to see if it ever saw the request.

ObjectClassError

This exception is raised when an object class is used that is not defined in the schema.

Others

Other exceptions may be raised by the Ruby/LDAP module, or by other subsystems. If you get one of these exceptions and think it should be wrapped, write me an email and let me know where it is and what you expected. For faster results, email a patch!

Putting it all together

Now that all of the components of ActiveLdap have been covered, it's time to put it all together! The rest of this section will show the steps to setup example user and group management scripts for use with the LDAP tree described above.

All of the scripts here are in the package's examples/ directory.

Setting up

Create directory for scripts.

```
% mkdir -p ldapadmin/objects
```

In ldapadmin/objects/ create the file user.rb:

```
require 'objects/group'

class User < ActiveLdap::Base
  ldap_mapping :dn_attribute => 'uid', :prefix => 'ou
  belongs_to :groups, :class_name => 'Group', :many =
end
```

In ldapadmin/objects/ create the file group.rb:

```
class Group < ActiveLdap::Base
  ldap_mapping :classes => ['top', 'posixGroup'], :pre
  has_many :members, :class_name => "User", :wrap => '
  has_many :primary_members, :class_name => 'User', :
end
```

Now, we can write some small scripts to do simple management tasks.

Creating LDAP entries

Now let's create a really dumb script for adding users – ldapadmin/useradd:

```
#!/usr/bin/ruby -W0

base = File.expand_path(File.join(File.dirname(__FILE_
$LOAD_PATH << File.join(base, "lib")
$LOAD_PATH << File.join(base, "examples")

require 'rubygems'
require 'active_ldap'
require 'objects/user'
require 'objects/group'

argv, opts, options = ActiveLdap::Command.parse_optio
  opts.banner += " USER_NAME CN UID"
end

if argv.size == 3
  name, cn, uid = argv
else
  $stderr.puts opts
  exit 1
end

pwb = Proc.new do |user|
  ActiveLdap::Command.read_password("[#{user}] Passwo
end

ActiveLdap::Base.setup_connection(:password_block => 
                                  :allow_anonymous => 

if User.exists?(name)
  $stderr.puts("User #{name} already exists.")
  exit 1
end

user = User.new(name)
user.add_class('shadowAccount')
user.cn = cn
user.uid_number = uid
user.gid_number = uid
user.home_directory = "/home/#{name}"
user.sn = "somesn"
unless user.save
  puts "failed"
  puts user.errors.full_messages
  exit 1
end
```

Managing LDAP entries

Now let’s create another dumb script for modifying users – ldapadmin/usermod:

```
#!/usr/bin/ruby -W0

base = File.expand_path(File.join(File.dirname(__FILE_
$LOAD_PATH << File.join(base, "lib")
$LOAD_PATH << File.join(base, "examples")

require 'rubygems'
require 'active_ldap'
require 'objects/user'
require 'objects/group'

argv, opts, options = ActiveLdap::Command.parse_optio
  opts.banner += " USER_NAME CN UID"
end

if argv.size == 3
  name, cn, uid = argv
else
  $stderr.puts opts
  exit 1
end

pwb = Proc.new do |user|
  ActiveLdap::Command.read_password("[#{user}] Passwo
end

ActiveLdap::Base.setup_connection(:password_block => 
                                  :allow_anonymous => 

unless User.exists?(name)
  $stderr.puts("User #{name} doesn't exist.")
  exit 1
end

user = User.find(name)
user.cn = cn
user.uid_number = uid
user.gid_number = uid
unless user.save
  puts "failed"
  puts user.errors.full_messages
  exit 1
end
```

Removing LDAP entries

Now let’s create more one for deleting users – ldapadmin/userdel:

```
#!/usr/bin/ruby -W0

base = File.expand_path(File.join(File.dirname(__FILE_
$LOAD_PATH << File.join(base, "lib")
$LOAD_PATH << File.join(base, "examples")

require 'rubygems'
require 'active_ldap'
require 'objects/user'
require 'objects/group'

argv, opts, options = ActiveLdap::Command.parse_optio
  opts.banner += " USER_NAME"
end

if argv.size == 1
  name = argv.shift
else
  $stderr.puts opts
  exit 1
end

pwb = Proc.new do |user|
  ActiveLdap::Command.read_password("[#{user}] Passwo
end

ActiveLdap::Base.setup_connection(:password_block =>
                                  :allow_anonymous =>

unless User.exists?(name)
  $stderr.puts("User #{name} doesn't exist.")
  exit 1
end

User.destroy(name)
```

Advanced Topics

Below are some situation tips and tricks to get the most out of ActiveLdap.

Binary data and other subtypes

Sometimes, you may want to store attributes with language specifiers, or perhaps in binary form. This is (finally!) fully supported. To do so, follow the examples below:

```
irb> user = User.new('drewry')
=> ...
# This adds a cn entry in lang-en and whatever the se
irb> user.cn = [ 'wad', {'lang-en' => ['wad', 'Will D
=> ...
irb> user.cn
=> ["wad", {"lang-en-us" => ["wad", "Will Drewry"]}]
# Now let's add a binary X.509 certificate (assume ob
irb> user.user_certificate = File.read('example.der')
=> ...
irb> user.save
```

So that’s a lot to take in. Here’s what is going on. I just set the LDAP object’s cn to “wad” and cn:lang-en-us to [“wad”, “Will Drewry”]. Anytime a LDAP subtype is required, you must encapsulate the data in a Hash.

But wait a minute, I just read in a binary certificate without wrapping it up. So any binary attribute *that requires ;binary subtyping* will automagically get wrapped in `{'binary' => value}` if you don’t do it. This keeps your #writes from breaking, and my code from crying. For correctness, I could have easily done the following:

```
irb> user.user_certificate = {'binary' => File.read('e
```

You should note that some binary data does not use the binary subtype all the time. One example is jpegPhoto. You can use it as jpegPhoto;binary or just as jpegPhoto. Since the schema dictates that it is a binary value, ActiveLdap will write it as binary, but the subtype will not be automatically appended as above. The use of the subtype on attributes like jpegPhoto is ultimately decided by the LDAP site policy and not by any programmatic means.

The only subtypes defined in LDAPv3 are lang-* and binary. These can be nested though:

```
irb> user.cn = [{'lang-ja' => {'binary' => 'some Japa
```

As I understand it, OpenLDAP does not support nested subtypes, but some documentation I’ve read suggests that Netscape’s LDAP server does. I only have access to OpenLDAP. If anyone tests this out, please let me know how it goes!

And that pretty much wraps up this section.

Further integration with your environment aka namespacing

If you want this to cleanly integrate into your system-wide Ruby include path, you should put your extension classes inside a custom module.

Example:

./myldap.rb:

```
require 'active_ldap'
require 'myldap/user'
require 'myldap/group'
module MyLDAP
end
```

./myldap/user.rb:

```
module MyLDAP
  class User < ActiveLdap::Base
    ldap_mapping :dn_attribute => 'uid', :prefix => '
    belongs_to :groups, :class_name => 'MyLDAP::Group
  end
end
```

./myldap/group.rb:

```
module MyLDAP
  class Group < ActiveLdap::Base
    ldap_mapping :classes => ['top', 'posixGroup'], :
    has_many :members, :class_name => 'MyLDAP::User',
    has_many :primary_members, :class_name => 'MyLDAP
  end
end
```

Now in your local applications, you can call

```
require 'myldap'

MyLDAP::Group.new('foo')
...
```

and everything should work well.

force array results for single values

Even though ActiveLdap attempts to maintain programmatic ease by returning Array values only. By specifying ‘true’ as an argument to any attribute method you will get back a Array if it is single value. Here’s an example:

```
irb> user = User.new('drewry')
=> ...
irb> user.cn(true)
=> ["Will Drewry"]
```

Dynamic attribute crawling

If you use tab completion in irb, you’ll notice that you /can/ tab complete the dynamic attribute methods. You can still see which methods are for attributes using Base#attribute_names:

```
irb> d = Group.new('develop')
=> ...
irb> d.attribute_names
=> ["gidNumber", "cn", "memberUid", "commonName", "de
```

Juggling multiple LDAP connections

In the same vein as the last tip, you can use multiple LDAP connections by per class as follows:

```
irb> anon_class = Class.new(Base)
=> ...
irb> anon_class.setup_connection
=> ...
irb> auth_class = Class.new(Base)
=> ...
irb> auth_class.setup_connection(:password_block => la
=> ...
```

This can be useful for doing authentication tests and other such tricks.

:try_sasl

If you have the Ruby/LDAP package with the SASL/GSSAPI patch from Ian MacDonald’s web site, you can use Kerberos to bind to your LDAP server. By default, :try_sasl is false.

Also note that you must be using OpenLDAP 2.1.29 or higher to use SASL/GSSAPI due to some bugs in older versions of OpenLDAP.

Don’t be afraid! [Internals]

Don’t be afraid to add more methods to the extensions classes and to experiment. That’s exactly how I ended up with this package. If you come up with something cool, please share it!

The internal structure of ActiveLdap::Base, and thus all its subclasses, is still in flux. I’ve tried to minimize the changes to the overall API, but the internals are still rough around the edges.

Where’s ldap_mapping data stored? How can I get to it?

When you call ldap_mapping, it overwrites several class methods inherited from Base:

- Base.base()
- Base.required_classes()
- Base.dn_attribute()

You can access these from custom class methods by calling MyClass.base(), or whatever. There are predefined instance methods for getting to these from any new instance methods you define:

- Base#base()
- Base#required_classes()
- Base#dn_attribute()

What else?

Well if you want to use the LDAP connection for anything, I’d suggest still calling Base.connection to get it. There really aren’t many other internals that need to be worried about. You could get the LDAP schema with Base.schema.

The only other useful tricks are dereferencing and accessing the stored data. Since LDAP attributes can have multiple names, e.g. cn or commonName, any methods you write might need to figure it out. I’d suggest just calling self[attribname] to get the value, but if that’s not good enough, you can call look up the stored name by

#to_real_attribute_name as follows:

```
irb> User.find(:first).instance_eval do
irb>   to_real_attribute_name('commonName')
irb> end
=> 'cn'
```

This tells you the name the attribute is stored in behind the scenes (@data). Again, self[attribname] should be enough for most extensions, but if not, it's probably safe to dabble here.

Also, if you like to look up all aliases for an attribute, you can call the following:

```
irb> User.schema.attribute_type 'cn', 'NAME'
=> ["cn", "commonName"]
```

This is discovered automagically from the LDAP server's schema.

Limitations

Speed

Currently, ActiveLdap could be faster. I have some recursive type checking going on which slows object creation down, and I'm sure there are many, many other places optimizations can be done. Feel free to send patches, or just hang in there until I can optimize away the slowness.

Feedback

Any and all feedback and patches are welcome. I am very excited about this package, and I'd like to see it prove helpful to more people than just myself.

FAMFAMFAM