Products.LDAPConnector Documentation

Release 1.0

Jens Vagelpohl

April 14, 2010

CONTENTS

`Products.LDAPConnector` provides an abstraction layer on top of *python-ldap*. It offers a connection object with simplified methods for inserting, modifying, searching and deleting records in the LDAP directory tree. Failover/redundancy can be achieved by supplying connection data for more than one LDAP server.

CHAPTER

ONE

NARRATIVE DOCUMENTATION

Narrative documentation explaining how to use `Products.LDAPConnector`.

1.1 Installation

You will need Python version 2.4 or better to run `Products.LDAPConnector`. Development of `Products.LDAPConnector` is done primarily under Python 2.6, so that version is recommended.

Warning: To successfully install `Products.LDAPConnector`, you will need *setuptools* installed on your Python system in order to run the `easy_install` command.

It is advisable to install `Products.LDAPConnector` into a *virtualenv* in order to obtain isolation from any "system" packages you've got installed in your Python version (and likewise, to prevent `Products.LDAPConnector` from globally installing versions of packages that are not compatible with your system Python).

After you've got the requisite dependencies installed, you may install `Products.LDAPConnector` into your Python environment using the following command:

```
$ easy_install Products.LDAPConnector
```

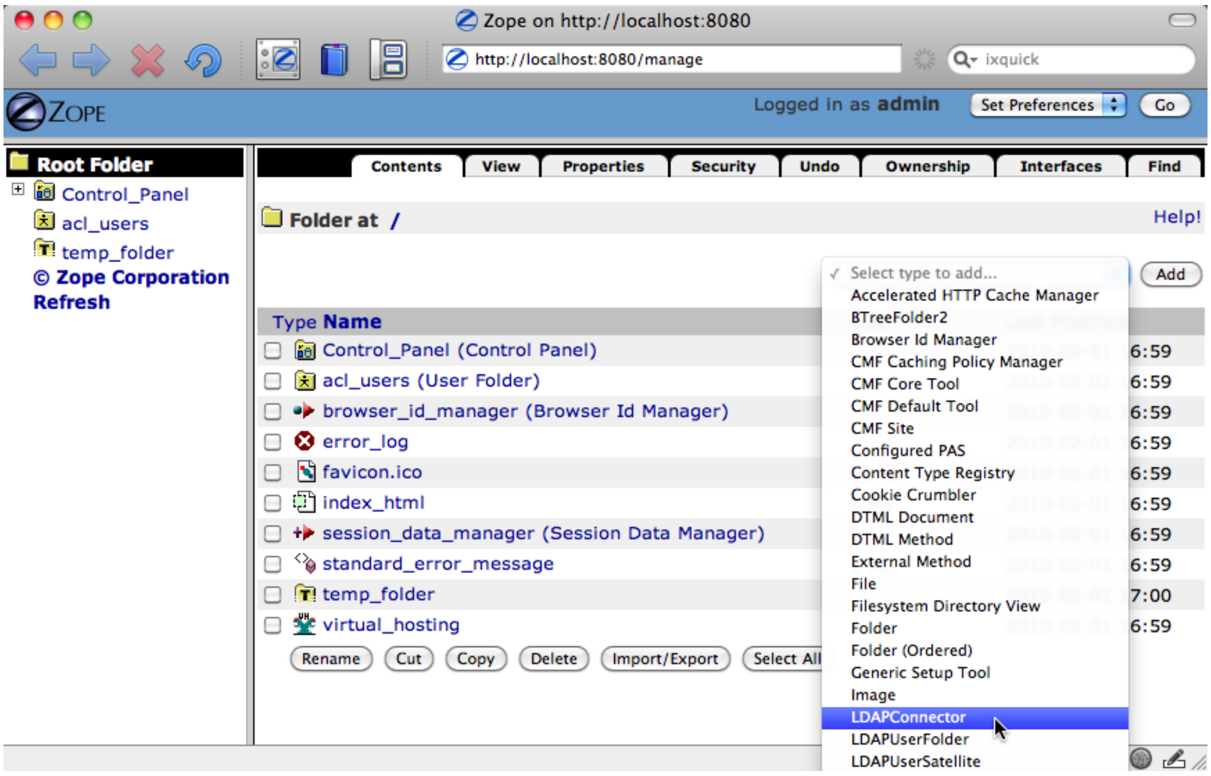
If you use `zc.buildout` you can add `Products.LDAPConnector` to the necessary `eggs` section to have it pulled in automatically.

When you `easy_install Products.LDAPConnector`, the *python-ldap* libraries are installed if they are not present.

1.2 Usage from the Zope ZMI

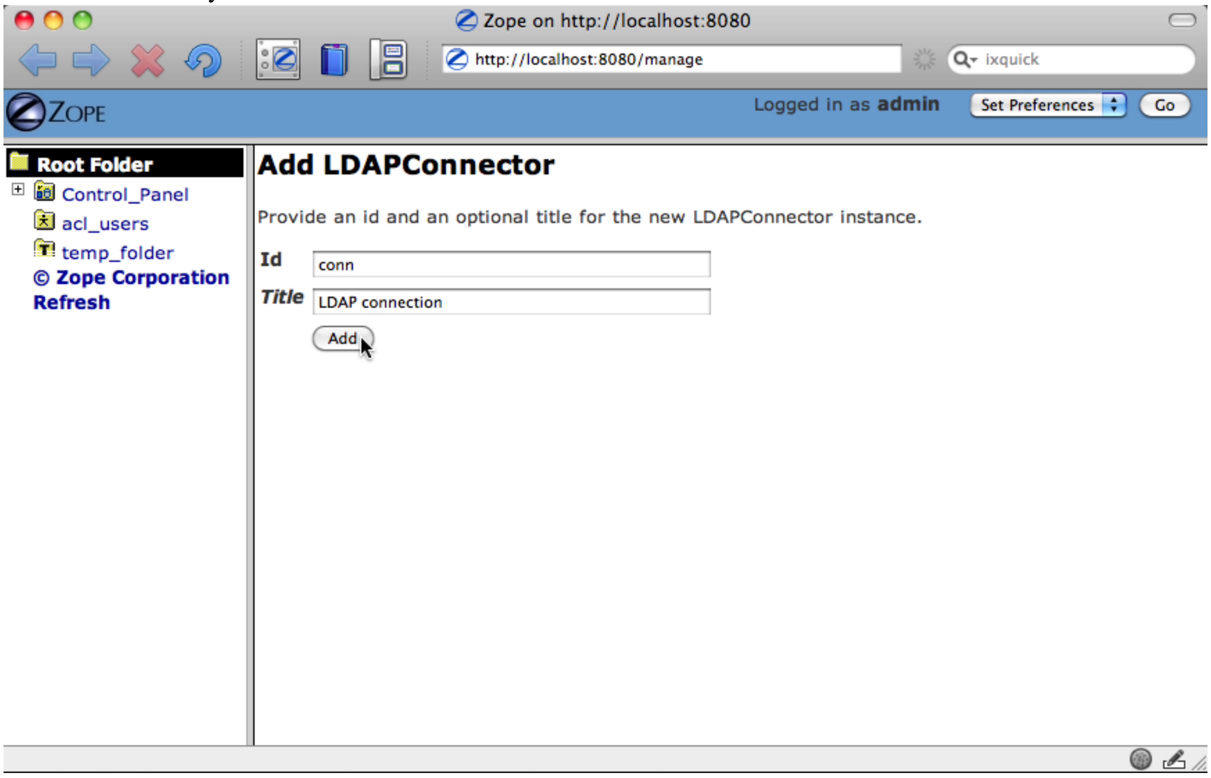
The following screen shots show how to use an LDAPConnector instance through the web in the *Zope ZMI*.

To create a new LDAPConnector instance, choose *LDAPConnector* from the drop-down list at the top right.



Type Name	
Control_Panel (Control Panel)	6:59
acl_users (User Folder)	6:59
browser_id_manager (Browser Id Manager)	6:59
error_log	6:59
favicon.ico	6:59
index_html	6:59
session_data_manager (Session Data Manager)	6:59
standard_error_message	6:59
temp_folder	7:00
virtual_hosting	6:59

On the initial view you can set the ID and title.



Add LDAPConnector

Provide an id and an optional title for the new LDAPConnector instance.

Id: conn

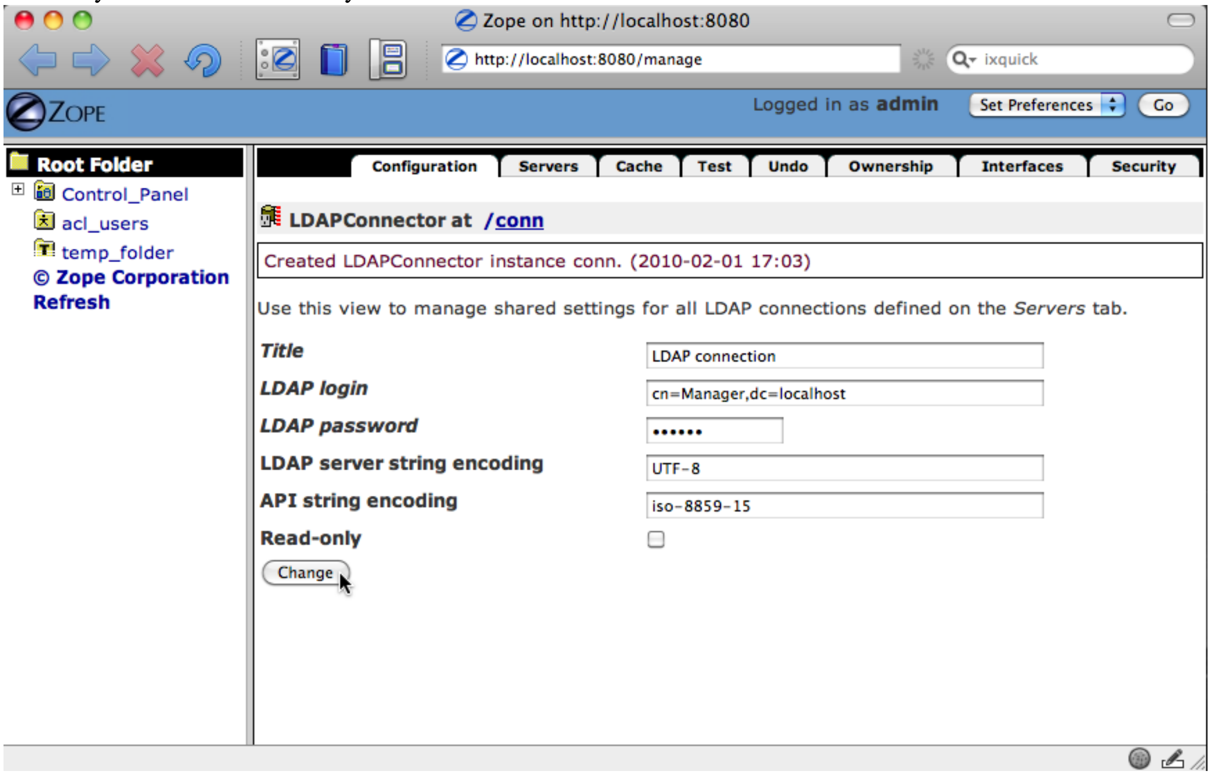
Title: LDAP connection

Once the instance is created you will end up on the `Configuration` tab where you can refine the configuration:

- The (optional) title can be chosen freely.
- The LDAP login and LDAP password fields allow you to specify LDAP credentials that will be used to authenticate to the LDAP servers set up in the next step. The same credentials will be used for all physical LDAP server connections set up on the `Servers` *ZMI* tab.
- LDAP servers store text values in specific text encodings, usually UTF-8. You need to specify the encoding

name in the LDAP server string encoding field.

- The API string encoding field is used to specify the text encoding applied to values returned by the LDAPConnector instance, as well as the expected encoding for values passed in using the instance's API. If you leave this field empty no encoding will be used, which means unencoded unicode.
- If you select the Read-only checkbox no writes to the LDAP server will be allowed.



LDAPConnector at /conn

Created LDAPConnector instance conn. (2010-02-01 17:03)

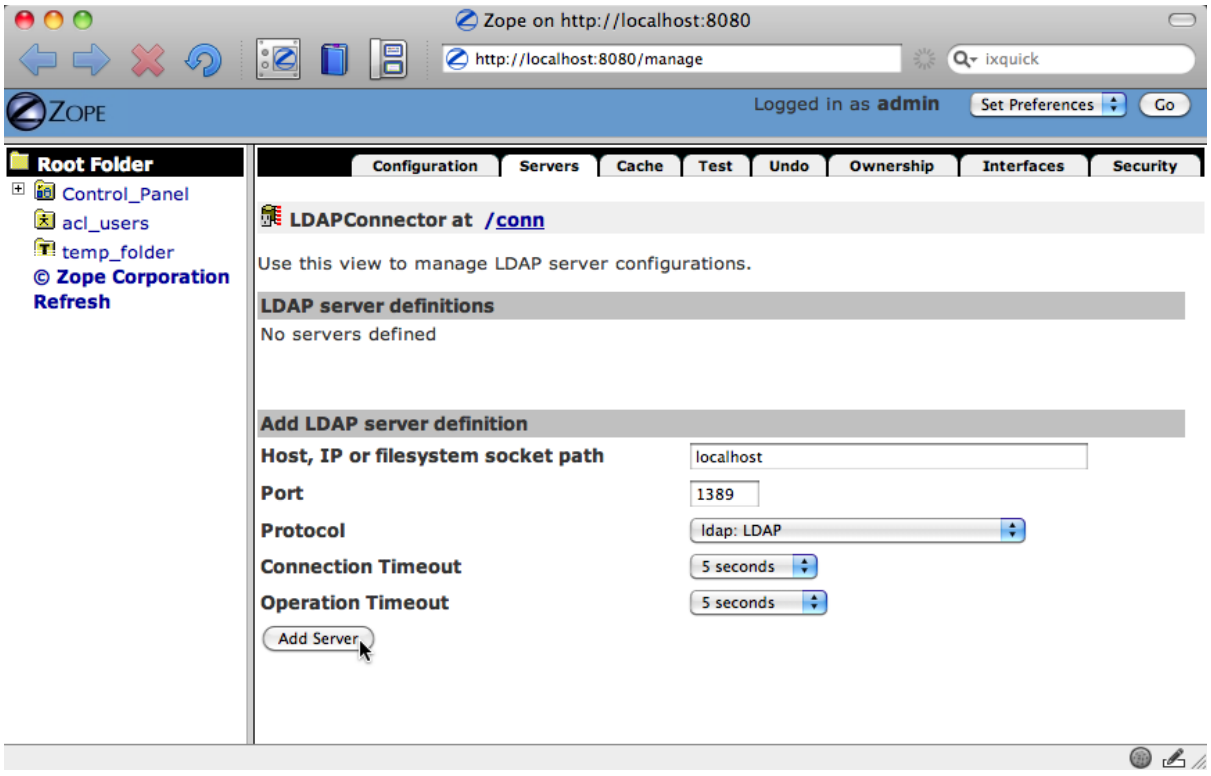
Use this view to manage shared settings for all LDAP connections defined on the *Servers* tab.

Title	LDAP connection
LDAP login	cn=Manager,dc=localhost
LDAP password	••••••
LDAP server string encoding	UTF-8
API string encoding	iso-8859-15
Read-only	☐

On the `Servers` tab you define physical server connections. By defining more than one server you can achieve redundancy, which means the LDAPConnector will be usable even if one server is no longer reachable or returns an error.

To set up a server connection you need to provide the following:

- a hostname, IP of a filesystem path for UNIX domain sockets
- a port number (this field is ignored when using UNIX domain sockets)
- the protocol to use, which can be `ldap` for unencrypted data transmission, `ldaps` for encrypted traffic using LDAP over SSL, `ldaptls` for negotiated encryption through the standard unencrypted port, or `ldapi` when using UNIX domain sockets
- a connection timeout value in seconds for the initial connection setup, after which a server is considered dead
- an operations timeout value in seconds to set a maximum allowable time for any operation, after which a server is considered dead



LDAPConnector at /conn

Use this view to manage LDAP server configurations.

LDAP server definitions

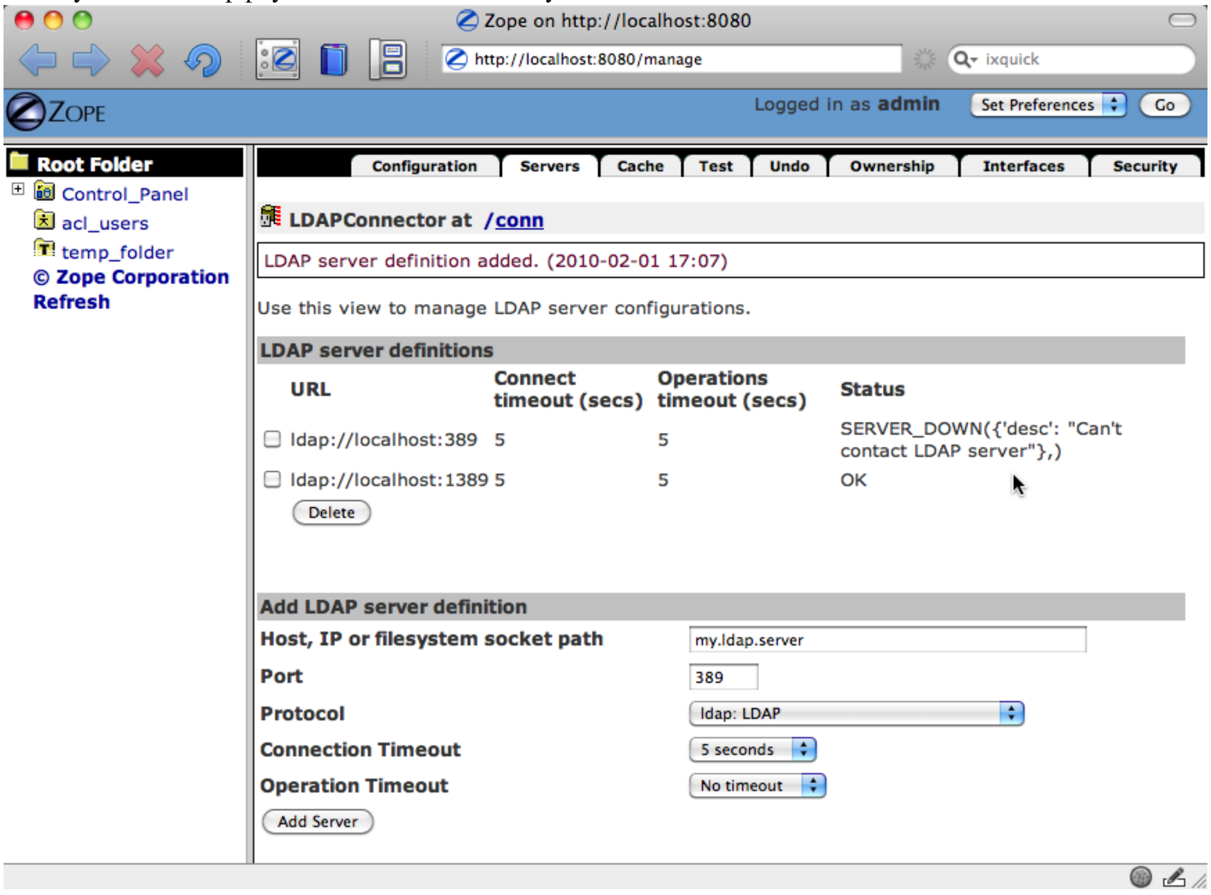
No servers defined

Add LDAP server definition

Host, IP or filesystem socket path	localhost
Port	1389
Protocol	ldap: LDAP
Connection Timeout	5 seconds
Operation Timeout	5 seconds

Add Server

When you have set up physical server connections you can see their status on the `Server` tab.



LDAPConnector at /conn

LDAP server definition added. (2010-02-01 17:07)

Use this view to manage LDAP server configurations.

LDAP server definitions

URL	Connect timeout (secs)	Operations timeout (secs)	Status
ldap://localhost:389	5	5	SERVER_DOWN({'desc': "Can't contact LDAP server"},)
ldap://localhost:1389	5	5	OK

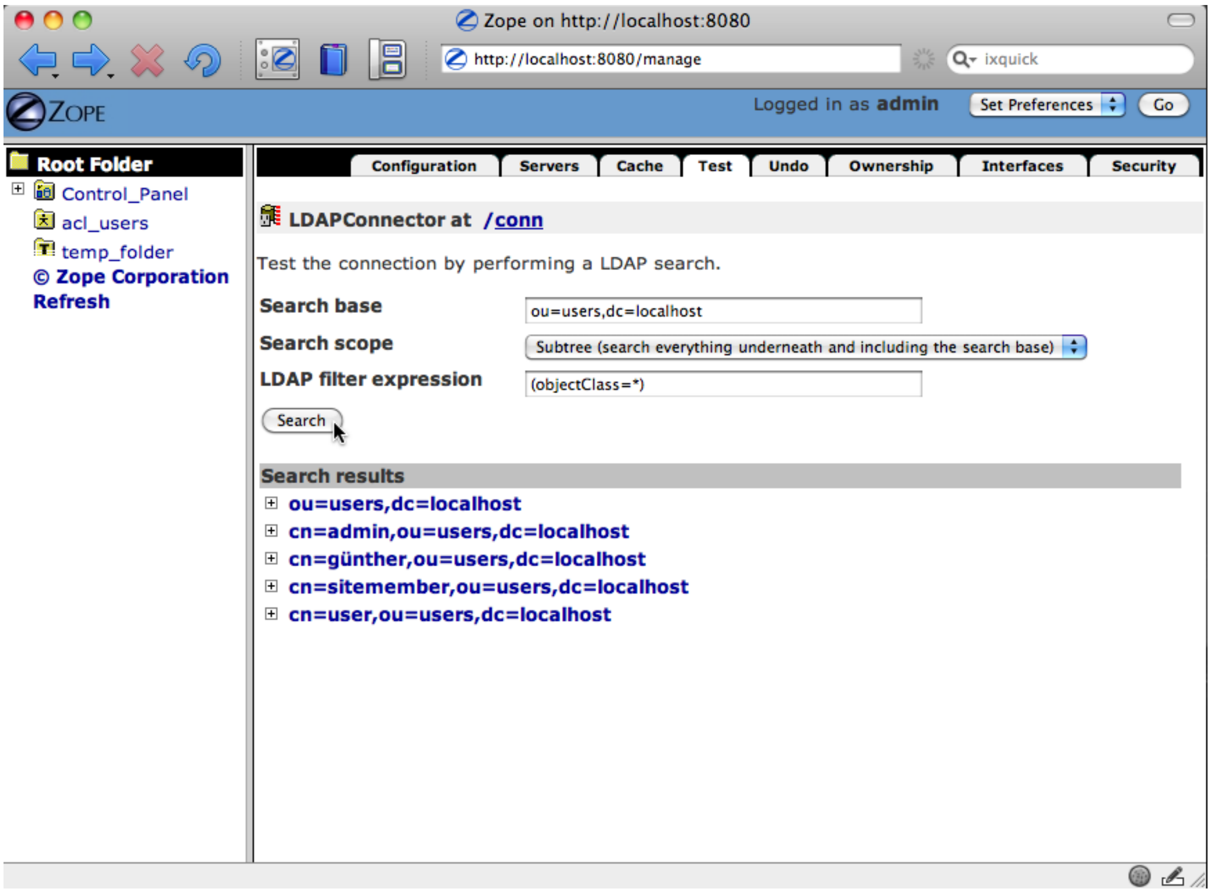
Delete

Add LDAP server definition

Host, IP or filesystem socket path	my.ldap.server
Port	389
Protocol	ldap: LDAP
Connection Timeout	5 seconds
Operation Timeout	No timeout

Add Server

With servers defined and at least one of them showing status OK you are ready to run a simple search test. The `Test` tab requires basic knowledge about your LDAP tree structure so you can pick a node for the Search base value. When executing the search you will see the results listed at the bottom.



LDAPConnector at /conn

Test the connection by performing a LDAP search.

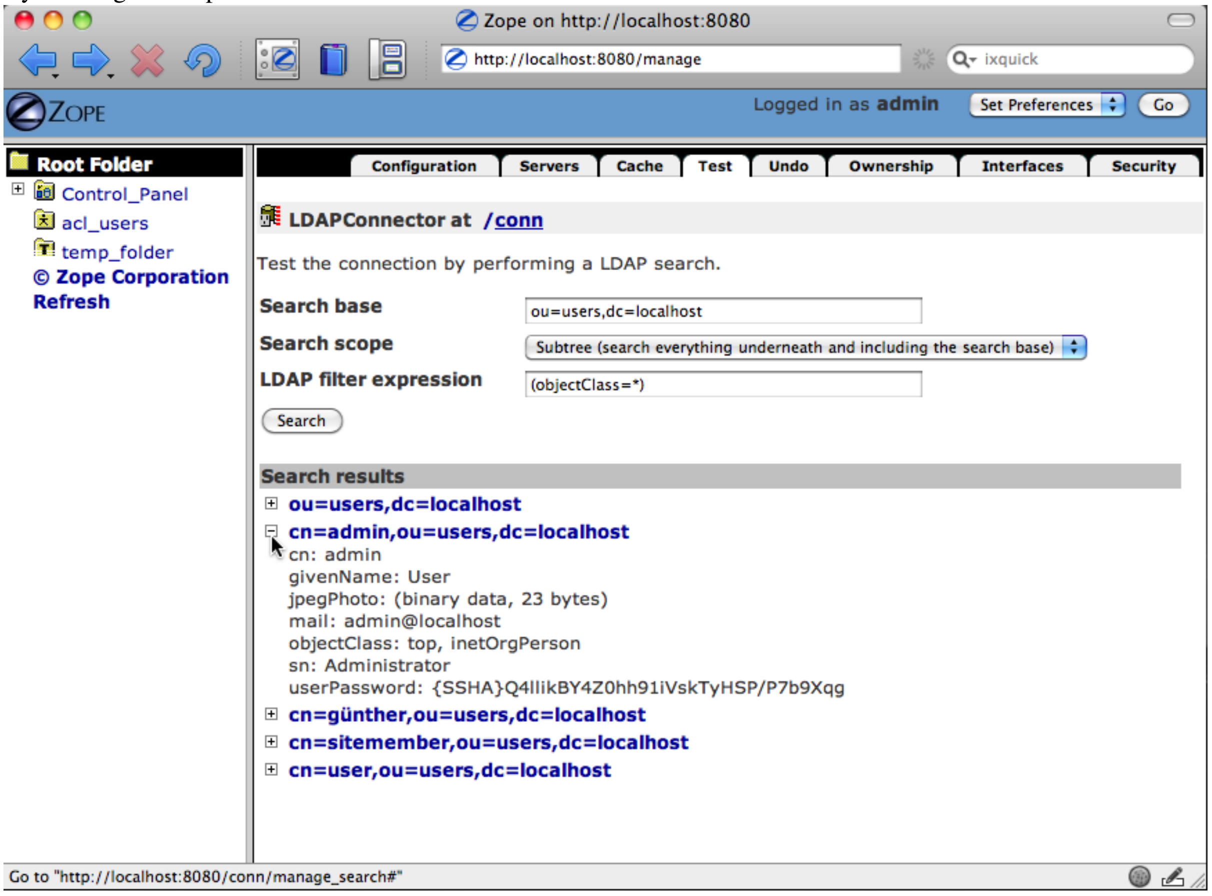
Search base	ou=users,dc=localhost
Search scope	Subtree (search everything underneath and including the search base)
LDAP filter expression	(objectClass=*)

Search

Search results

```
⊞ ou=users,dc=localhost
⊞ cn=admin,ou=users,dc=localhost
⊞ cn=günther,ou=users,dc=localhost
⊞ cn=sitemember,ou=users,dc=localhost
⊞ cn=user,ou=users,dc=localhost
```

By clicking on the plus icon or the DN the records can be examined in detail.



LDAPConnector at /conn

Test the connection by performing a LDAP search.

Search base	ou=users,dc=localhost
Search scope	Subtree (search everything underneath and including the search base)
LDAP filter expression	(objectClass=*)

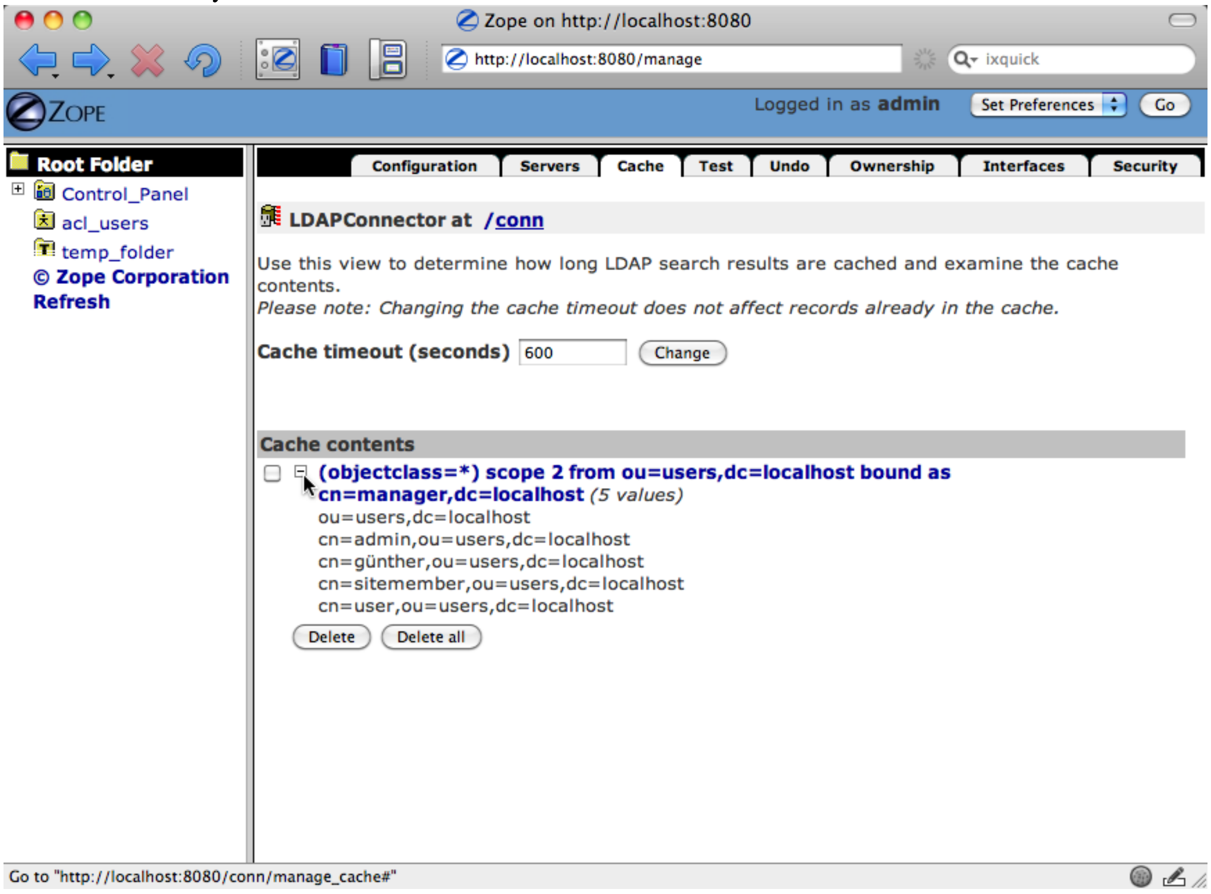
Search

Search results

```
⊞ ou=users,dc=localhost
⊟ cn=admin,ou=users,dc=localhost
  cn: admin
  givenName: User
  jpegPhoto: (binary data, 23 bytes)
  mail: admin@localhost
  objectClass: top, inetOrgPerson
  sn: Administrator
  userPassword: {SSHA}Q4lIikBY4Z0hh91iVskTyHSP/P7b9Xqg
⊞ cn=günther,ou=users,dc=localhost
⊞ cn=sitemember,ou=users,dc=localhost
⊞ cn=user,ou=users,dc=localhost
```

Search records are cached. On the `Cache` tab you can set the number of seconds each search and its results are cached. You can also view what's in the cache. From here, you can delete specific cache entries or flush the whole cache.

The cache works as a negative cache as well. Searches that return error messages or no results at all will be cached to avoid unnecessary work.



LDAPConnector at /conn

Use this view to determine how long LDAP search results are cached and examine the cache contents.
Please note: Changing the cache timeout does not affect records already in the cache.

Cache timeout (seconds) 600

Cache contents

(objectclass=*) scope 2 from ou=users,dc=localhost bound as cn=manager,dc=localhost *(5 values)*

```
ou=users,dc=localhost
cn=admin,ou=users,dc=localhost
cn=günther,ou=users,dc=localhost
cn=sitemember,ou=users,dc=localhost
cn=user,ou=users,dc=localhost
```

1.3 Usage from Python

These samples assume that you have a LDAPConnector instance set up already with ID `conn`.

Adding a server definition:

```
>>> conn.addServer('localhost', '1389', 'ldap')
```

To work with the connection object you need to make sure that a LDAP server is available on the provided host and port.

Now we will search for a record that does not yet exist, then add the missing record and find it when searching again:

```
>>> conn.search('ou=users,dc=localhost', fltr='(cn=testing)')
{'exception': '', 'results': [], 'size': 0}
>>> data = { 'objectClass': ['top', 'inetOrgPerson']
...        , 'cn': 'testing'
...        , 'sn': 'Lastname'
...        , 'givenName': 'Firstname'
...        , 'mail': 'test@test.com'
...        , 'userPassword': '5ecret'
...        }
```

```
>>> conn.insert('ou=users,dc=localhost', 'cn=testing', attrs=data, bind_dn='cn=Manager,dc=localh
>>> conn.search('ou=users,dc=localhost', fltr='(cn=testing)')
{'exception': '', 'results': [{'dn': 'cn=testing,ou=users,dc=localhost', 'cn': ['testing'], 'obj
```

We can edit an existing record:

```
>>> changes = {'givenName': 'John', 'sn': 'Doe'}
>>> conn.modify('cn=testing,ou=users,dc=localhost', attrs=changes, bind_dn='cn=Manager,dc=localho
>>> conn.search('ou=users,dc=localhost', fltr='(cn=testing)')
{'exception': '', 'results': [{'dn': 'cn=testing,ou=users,dc=localhost', 'cn': ['testing'], 'obje
```

As the last step, we will delete our testing record:

```
>>> conn.delete('cn=testing,ou=users,dc=localhost', bind_dn='cn=Manager,dc=localhost', bind_pwd='
>>> conn.search('ou=users,dc=localhost', fltr='(cn=testing)')
{'exception': '', 'results': [], 'size': 0}
```

The *Interfaces* page contains more information about the connection APIs.

1.4 String encoding issues

LDAP servers expect values sent to them in specific string encodings. Standards-compliant LDAP servers use UTF-8. They use the same encoding for values returned e.g. by a search. This server-side encoding may not be convenient for communicating with the `Products.LDAPConnector` API itself. For this reason the server-side encoding and API encoding can be set individually on connection instances using the attributes `ldap_encoding` and `api_encoding`, respectively. The connection instance handles all string encoding transparently.

By default, instances use UTF-8 as `ldap_encoding` and ISO-8859-1 (Latin-1) as `api_encoding`. You can assign any valid Python codec name to these attributes. Assigning an empty value or None means that unencoded unicode strings are used.

If you receive error messages and tracebacks for either `UnicodeDecodeError` or `UnicodeEncodeError` while searching for records on the *ZMI* `Test` tab or while displaying LDAPConnector search results in your own web application using Zope Page Templates, you have several places to look at:

- Make sure the LDAPConnector `ldap_encoding` value, visible on the *ZMI* `Configuration` tab as *LDAP server string encoding*, is set to the encoding required by your LDAP server. For most servers this will be UTF-8. With *Active Directory* this may differ.
- Check the text encoding used by your web application. It will usually be something like `iso-8859-1` or `utf-8`. Make sure it matches the LDAPConnector `api_encoding` value, which is set on the *ZMI* `Configuration` tab as *API string encoding*. If you leave this field empty unencoded unicode is expected by the API and will be returned by it.
- If your browser does not send along its preferred character encoding when requesting data from your server (request header `HTTP_ACCEPT_CHARSET`) Zope may pick the wrong text encoding. Safari-based browsers like Safari or Omniweb show this behavior. You can influence which encoding gets picked by overriding a *ZCML* registration in your site's configuration. To use the encoding defined as `management_page_charset` in your site, add the following to your site configuration:

  ```
  <utility
    provides="Products.PageTemplates.interfaces.IUnicodeEncodingConflictResolver"
    component="Products.PageTemplates.unicodeconflictresolver.StrictUnicodeEncodingConflictResc
    />
  ```

1.5 Development

1.5.1 Getting the source code

The source code is maintained in the Dataflake Subversion repository at http://svn.dataflake.org. To check out the trunk:

```
svn co http://svn.dataflake.org/svn/Products.LDAPConnector/trunk/
```

You can also browse the code online at http://svn.dataflake.org/viewvc/Products.LDAPConnector.

When using setuptools or zc.buildout you can use the following URL to retrieve the latest development code as Python egg:

```
http://svn.dataflake.org/svn/Products.LDAPConnector/trunk#egg=Products.LDAPConnector
```

1.5.2 Bug tracker

For bug reports, suggestions or questions please use the dataflake bug tracker at https://bugs.launchpad.net/products.ldapconnector.

1.5.3 Setting up a development sandbox and testing

Once you've obtained a source checkout, you can follow these instructions to perform various development tasks. All development requires that you run the buildout from the package root directory:

```
$ python bootstrap.py
$ bin/buildout
```

Once you have a buildout, the tests can be run as follows:

```
$ bin/test
```

1.5.4 Building the documentation

The Sphinx documentation is built by doing the following from the directory containing setup.py:

```
$ cd docs
$ make html
```

1.5.5 Making a release

The first thing to do when making a release is to check that the ReST to be uploaded to PyPI is valid:

```
$ bin/docpy setup.py --long-description | bin/rst2 html \
  --link-stylesheet \
  --stylesheet=http://www.python.org/styles/styles.css > build/desc.html
```

Once you're certain everything is as it should be, the following will build the distribution, upload it to PyPI, register the metadata with PyPI and upload the Sphinx documentation to PyPI:

```
$ bin/buildout -o
$ bin/docpy setup.py sdist register upload upload_sphinx --upload-dir=docs/_build/html
```

The `bin/buildout` will make sure the correct package information is used.

1.6 Changelog for Products.LDAPConnector

1.6.1 1.0 (2010-04-14)

- initial release

CHAPTER

TWO

API DOCUMENTATION

API documentation for `Products.LDAPConnector`.

2.1 Interfaces

Instances of `Products.LDAPConnector` derive from the `dataflake.ldapconnection.connection` module's `LDAPConnection` class and implement the interface `dataflake.ldapconnection.interfaces.ILDAConne`
They mix in *Zope* persistence to allow storage in the *ZODB* object database.

`interface dataflake.ldapconnection.interfaces.ILDAPConnection`
ILDAPConnection interface

ILDAPConnection instances provide a simplified way to talk to a LDAP server. They allow defining one or more server connections for automatic failover in case one LDAP server becomes unavailable.

`insert`(*base, rdn, attrs=None, bind_dn=None, bind_pwd=None*)
Insert a new record

The record will be inserted at *base* with the new RDN *rdn*. *attrs* is expected to be a key:value mapping where the value may be a string or a sequence of strings. Multiple values may be expressed as a single string if the values are semicolon-delimited. Values can be marked as binary values, meaning they are not encoded in the encoding specified as the server encoding before being sent to the LDAP server, by appending ';binary' to the key.

In order to perform the operation using credentials other than the credentials configured on the instance a DN and password may be passed in.

`addServer`(*host, port, protocol, conn_timeout=-1, op_timeout=-1*)
Add a server definition

protocol can be any one of `ldap` (unencrypted traffic), `ldaps` (encrypted traffic to a separate port), `ldaptls` (sets up encrypted traffic on the normal unencrypted port), or `ldapi` (trafic through a UNIX domain socket on the file system).

The *conn_timeout* argument defines the number of seconds to wait until a new connection attempt is considered failed, which means the next server is tried if it has been defined. -1 means "wait indefinitely",

The *op_timeout* argument defines the number of seconds to wait until a LDAP server operation is considered failed, which means the next server is tried if it has been defined. -1 means "wait indefinitely".

If a server definition with a host, port and protocol that matches an existing server definition is added, the new values will replace the existing definition.

`modify`(*dn, mod_type=None, attrs=None, bind_dn=None, bind_pwd=None*)
Modify the record specified by the given DN

mod_type is one of the LDAP modification types as declared by the *python-ldap*-module, such as *ldap.MOD_ADD*, PUrl(urlscheme=protocol, hostport=hostport) provided, the modification type is guessed by comparing the current record with the *attrs* mapping passed in.

attrs is expected to be a key:value mapping where the value may be a string or a sequence of strings. Multiple values may be expressed as a single string if the values are semicolon-delimited. Values can be marked as binary values, meaning they are not encoded as UTF-8 before sending the to the LDAP server, by appending ';binary' to the key.

In order to perform the operation using credentials other than the credentials configured on the instance a DN and password may be passed in.

`search`(*base, scope=2, fltr='(objectClass=*)', attrs=None, convert_filter=True, bind_dn=None, bind_pwd=None*)

Perform a LDAP search

The search *base* is the point in the tree to search from. *scope* defines how to search and must be one of the scopes defined by the *python-ldap* module (*ldap.SCOPE_BASE*, *ldap.SCOPE_ONELEVEL* or *ldap.SCOPE_SUBTREE*). By default, *ldap.SCOPE_SUBTREE* is used. What to search for is described by the *filter* argument, which must be a valid LDAP search filter string. If only certain record attributes should be returned, they can be specified in the *attrs* sequence.

If the search raised no errors, a mapping with the following keys is returned:

- results: A sequence of mappings representing a matching record
- size: The number of matching records

The results sequence itself contains mappings that have a *dn* key containing the full distinguished name of the record, and key/values representing the records' data as returned by the LDAP server.

In order to perform the operation using credentials other than the credentials configured on the instance a DN and password may be passed in.

`removeServer`(*host, port, protocol*)

Remove a server definition

Please note: I you remove the server definition of a server that is currently being used, that connection will continue to be used until it fails or until the Python process is restarted.

`connect`(*bind_dn=None, bind_pwd=None*)

Return a working LDAP server connection

If no DN or password for binding to the LDAP server are passed in, the DN and password configured into the LDAP connection instance are used.

The connection is cached and will be re-used. Since a bind operation is forced every time the method can be used to re-bind the cached connection with new credentials.

This method returns an instance of the underlying *python-ldap* connection class. It does not need to be called explicitly, all other operations call it implicitly.

Raises RuntimeError if no server definitions are available. If all defined server connections fail the LDAP exception thrown by the last attempted connection is re-raised.

`delete`(*dn, bind_dn=None, bind_pwd=None*)

Delete the record specified by the given DN

In order to perform the operation using credentials other than the credentials configured on the instance a DN and password may be passed in.

CHAPTER

THREE

SUPPORT

If you need commercial support for this software package, please contact zetwork GmbH at http://www.zetwork.com.

CHAPTER

FOUR

INDICES AND TABLES

- *Index*
- *Search Page*
- *Glossary*

INDEX