Mailman CLI Documentation

Release 1

Rajeev S

September 26, 2014

Contents

This package is called `mailman.client`.

CHAPTER 1

mailman.client

The `mailman.client` library provides official Python bindings for the GNU Mailman 3 REST API.

Note that the test suite current requires that a Mailman 3 server be running. It should be running using a dummy or throw-away database, as this will make changes to the running system. TBD: mock the real Mailman engine so that it is not necessary in order to run these tests.

1.1 Requirements

`mailman.client` requires Python 2.6 or newer.

1.2 Project details

You may download the latest version of the package from the Python Cheese Shop or from Launchpad.

You can also install it via `easy_install` or `pip`.:

```
% sudo easy_install mailman.client
% sudo pip install mailman.client
```

See the Launchpad project page for access to the Bazaar branch, bug report, etc.

1.3 Documentation

A simple guide to using the library is available within this package, in the form of doctests. The manual is also available online in the Cheeseshop at:

http://package.python.org/mailman.client

1.4 Acknowledgements

Many thanks to Florian Fuchs for his contribution of an initial REST client.

1.5 Table of Contents

1.5.1 Mailman REST client

```
>>> import os
>>> import time
>>> import smtplib
>>> import subprocess

>>> from mock import patch
```

This is the official Python bindings for the GNU Mailman REST API. In order to talk to Mailman, the engine's REST server must be running. You begin by instantiating a client object to access the root of the REST hierarchy, providing it the base URL, user name and password (for Basic Auth).

```
>>> from mailmanclient import Client
>>> client = Client('http://localhost:9001/3.0', 'restadmin', 'restpass')
```

We can retrieve basic information about the server.

```
>>> dump(client.system)
http_etag: "..."
mailman_version: GNU Mailman 3.0... (...)
python_version: ...
self_link: http://localhost:9001/3.0/system
```

To start with, there are no known mailing lists.

```
>>> client.lists
[]
```

Domains

Before new mailing lists can be added, the domain that the list will live in must be added. By default, there are no known domains.

```
>>> client.domains
[]
```

It's easy to create a new domain; when you do, a proxy object for that domain is returned.

```
>>> example_dot_com = client.create_domain('example.com',
...     contact_address='admin@example.com')
>>> example_dot_com
<Domain "example.com">
>>> print example_dot_com.base_url
http://example.com
>>> print example_dot_com.contact_address
admin@example.com
>>> print example_dot_com.description
None
>>> print example_dot_com.mail_host
example.com
>>> print example_dot_com.url_host
example.com
```

You can also get an existing domain independently using its mail host.

```
>>> example = client.get_domain('example.com')
>>> example
<Domain "example.com">
>>> print example_dot_com.base_url
http://example.com
```

Additionally you can get an existing domain using its web host.

```
>>> example = client.get_domain(web_host='http://example.com')
>>> example
<Domain "example.com">
>>> print example_dot_com.base_url
http://example.com
```

But you cannot retrieve a non-existent domain.

```
>>> client.get_domain('example.org')
Traceback (most recent call last):
...
HTTPError: HTTP Error 404: 404 Not Found
```

After creating a few more domains, we can print the list of all domains.

```
>>> client.create_domain('example.net')
<Domain "example.net">
>>> client.create_domain('example.org')
<Domain "example.org">
>>> for mail_host in client.domains:
...     print mail_host
<Domain "example.com">
<Domain "example.net">
<Domain "example.org">
```

Also, domain can be deleted.

```
>>> client.delete_domain('example.org')
>>> for mail_host in client.domains:
...     print mail_host
<Domain "example.com">
<Domain "example.net">
```

Mailing lists

Once you have a domain, you can create mailing lists in that domain.

```
>>> test_one = example.create_list('test-one')
>>> test_one
<List "test-one@example.com">
>>> print test_one.fqdn_listname
test-one@example.com
>>> print test_one.mail_host
example.com
>>> print test_one.list_name
test-one
>>> print test_one.display_name
Test-one
```

You can also retrieve the mailing list after the fact.

```
>>> my_list = client.get_list('test-one@example.com')
>>> my_list
<List "test-one@example.com">
```

And you can print all the known mailing lists.

```
>>> example.create_list('test-two')
<List "test-two@example.com">
>>> domain = client.get_domain('example.net')
>>> domain.create_list('test-three')
<List "test-three@example.net">
>>> example.create_list('test-three')
<List "test-three@example.com">

>>> for mlist in client.lists:
...     print mlist
<List "test-one@example.com">
<List "test-two@example.com">
<List "test-three@example.net">
<List "test-three@example.com">
```

List results can be retrieved as pages:

```
>>> page = client.get_list_page(count=2, page=1)
>>> page.nr
1
>>> len(page)
2
>>> for m_list in page:
...     print m_list
<List "test-one@example.com">
<List "test-two@example.com">
>>> page = page.next
>>> page.nr
2
>>> for m_list in page:
...     print m_list
<List "test-three@example.net">
<List "test-three@example.com">
```

If you only want to know all lists for a specific domain, use the domain object.

```
>>> for mlist in example.lists:
...     print mlist
<List "test-one@example.com">
<List "test-three@example.com">
<List "test-two@example.com">
```

You can use a list instance to delete the list.

```
>>> test_three = client.get_list('test-three@example.net')
>>> test_three.delete()
```

You can also delete a list using the client instance's delete_list method.

```
>>> client.delete_list('test-three@example.com')

>>> for mlist in client.lists:
...     print mlist
```

```
<List "test-one@example.com">
<List "test-two@example.com">
```

Membership

Email addresses can subscribe to existing mailing lists, becoming members of that list. The address is a unique id for a specific user in the system, and a member is a user that is subscribed to a mailing list. Email addresses need not be pre-registered, though the auto-registered user will be unique for each email address.

The system starts out with no members.

```
>>> client.members
[]
```

New members can be easily added; users are automatically registered.

```
>>> test_two = client.get_list('test-two@example.com')

>>> test_one.subscribe('anna@example.com', 'Anna')
<Member "anna@example.com" on "test-one.example.com">
>>> test_one.subscribe('bill@example.com', 'Bill')
<Member "bill@example.com" on "test-one.example.com">
>>> test_two.subscribe('anna@example.com')
<Member "anna@example.com" on "test-two.example.com">
>>> test_two.subscribe('cris@example.com', 'Cris')
<Member "cris@example.com" on "test-two.example.com">
```

We can retrieve all known memberships. These are sorted first by mailing list name, then by email address.

```
>>> for member in client.members:
...     print member
<Member "anna@example.com" on "test-one.example.com">
<Member "bill@example.com" on "test-one.example.com">
<Member "anna@example.com" on "test-two.example.com">
<Member "cris@example.com" on "test-two.example.com">
```

We can also view the memberships for a single mailing list.

```
>>> for member in test_one.members:
...     print member
<Member "anna@example.com" on "test-one.example.com">
<Member "bill@example.com" on "test-one.example.com">
```

Membership lists can be paginated, to recieve only a part of the result.

```
>>> page = client.get_member_page(count=2, page=1)
>>> page.nr
1
>>> for member in page:
...     print member
<Member "anna@example.com" on "test-one.example.com">
<Member "bill@example.com" on "test-one.example.com">

>>> page = page.next
>>> page.nr
2
>>> for member in page:
...     print member
```

```
<Member "anna@example.com" on "test-two.example.com">
<Member "cris@example.com" on "test-two.example.com">

>>> page = test_one.get_member_page(count=1, page=1)
>>> page.nr
1
>>> for member in page:
...     print member
<Member "anna@example.com" on "test-one.example.com">
>>> page = page.next
>>> page.nr
2
>>> for member in page:
...     print member
<Member "bill@example.com" on "test-one.example.com">
```

We can get a single membership too.

```
>>> cris_test_two = test_two.get_member('cris@example.com')
>>> cris_test_two
<Member "cris@example.com" on "test-two.example.com">
>>> print cris_test_two.role
member
```

A membership has preferences.

```
>>> prefs = cris_test_two.preferences
>>> print prefs['delivery_mode']
regular
>>> print prefs['acknowledge_posts']
None
>>> print prefs['delivery_status']
None
>>> print prefs['hide_address']
None
>>> print prefs['preferred_language']
None
>>> print prefs['receive_list_copy']
None
>>> print prefs['receive_own_postings']
None
```

The membership object's `user` attribute will return a User object:

```
>>> cris_test_two.user
<User "Cris" (...)>
```

If you use an address which is not a member of test_two *ValueError* is raised:

```
>>> test_two.unsubscribe('nomember@example.com')
Traceback (most recent call last):
...
ValueError: nomember@example.com is not a member address of test-two@example.com
```

After a while, Anna decides to unsubscribe from the Test One mailing list, though she keeps her Test Two membership active.

```
>>> test_one.unsubscribe('anna@example.com')
>>> for member in client.members:
...     print member
```

```
<Member "bill@example.com" on "test-one.example.com">
<Member "anna@example.com" on "test-two.example.com">
<Member "cris@example.com" on "test-two.example.com">
```

A little later, Cris decides to unsubscribe from the Test Two mailing list.

```
>>> cris_test_two.unsubscribe()
>>> for member in client.members:
...     print member
<Member "bill@example.com" on "test-one.example.com">
<Member "anna@example.com" on "test-two.example.com">
```

If you try to unsubscribe an address which is not a member address *ValueError* is raised:

```
>>> test_one.unsubscribe('nomember@example.com')
Traceback (most recent call last):
...
ValueError: nomember@example.com is not a member address of test-one@example.com
```

Users

Users are people with one or more list memberhips. To get a list of all users, access the clients user property.

```
>>> for user in client.users:
...     print user
<User "..." (...)>
<User "..." (...)>
<User "..." (...)>
```

The users can also be paginated:

```
>>> page = client.get_user_page(count=2, page=1)
>>> page.nr
1

>>> for user in page:
...     print user
<User "Anna" (...)>
<User "Bill" (...)>
```

You can get the next or previous pages without calling `get_userpage` again.

```
>>> page = page.next
>>> page.nr
2

>>> for user in page:
...     print user
<User "Cris" (...)>

>>> page = page.previous
>>> page.nr
1

>>> for user in page:
...     print user
<User "Anna" (...)>
<User "Bill" (...)>
```

A single user can be retrieved using their email address.

```
>>> cris = client.get_user('cris@example.com')
>>> print cris.display_name
Cris
```

Every user has a list of one or more addresses.

```
>>> for address in cris.addresses:
...     print address
...     print address.display_name
...     print address.registered_on
cris@example.com
Cris
...
```

Addresses can be accessed directly:

```
>>> address = client.get_address('cris@example.com')
>>> print address
cris@example.com
>>> print address.display_name
Cris
```

The address has not been verified:

```
>>> print address.verified_on is None
True
```

But that can be done via the address object:

```
>>> address.verify()
>>> address.verified_on is None
False
```

It can also be unverified:

```
>>> address.unverify()
>>> address.verified_on is None
True
```

Users can be added using `create_user`. The display_name is optional:

```
>>> client.create_user(email='ler@primus.org',
...                    password='somepass',
...                    display_name='Ler')
<User "Ler" (...)>
>>> ler = client.get_user('ler@primus.org')
>>> print ler.password
$...
>>> print ler.display_name
Ler
```

User attributes can be changed through assignment, but you need to call the object's `save` method to store the changes in the mailman core database.

```
>>> ler.display_name = 'Sir Ler'
>>> ler.save()
>>> ler = client.get_user('ler@primus.org')
>>> print ler.display_name
Sir Ler
```

Passwords can be changed as well:

```
>>> old_pwd = ler.password
>>> ler.password = 'easy'
>>> old_pwd == ler.password
True
>>> ler.save()
>>> old_pwd == ler.password
False
```

User Subscriptions

A User's subscriptions can be access through their `subscriptions` property.

```
>>> bill = client.get_user('bill@example.com')
>>> for subscription in bill.subscriptions:
...     print subscription
<Member "bill@example.com" on "test-one.example.com">
```

If all you need are the list ids of all mailing lists a user is subscribed to, you can use the `subscription_list_ids` property.

```
>>> for list_id in bill.subscription_list_ids:
...     print list_id
test-one.example.com
```

List Settings

We can get all list settings via a lists settings attribute. A proxy object for the settings is returned which behaves much like a dictionary.

```
>>> settings = test_one.settings
>>> len(settings)
48

>>> for attr in sorted(settings):
...     print attr + ': ' + str(settings[attr])
acceptable_aliases: []
...
welcome_message_uri: mailman:///welcome.txt

>>> print settings['display_name']
Test-one
```

We can access all valid list settings as attributes.

```
>>> print settings['fqdn_listname']
test-one@example.com
>>> print settings['description']

>>> settings['description'] = 'A very meaningful description.'
>>> settings['display_name'] = 'Test Numero Uno'

>>> settings.save()
```

```
>>> settings_new = test_one.settings
>>> print settings_new['description']
A very meaningful description.
>>> print settings_new['display_name']
Test Numero Uno
```

The settings object also supports the *get* method of usual Python dictionaries:

```
>>> print settings_new.get('OhNoIForgotTheKey', 'HowGoodIPlacedOneUnderTheDoormat')
HowGoodIPlacedOneUnderTheDoormat
```

Preferences

Preferences can be accessed and set for users, members and addresses.

By default, preferences are not set and fall back to the global system preferences. They're read-only and can be accessed through the client object.

```
>>> global_prefs = client.preferences
>>> print global_prefs['acknowledge_posts']
False
>>> print global_prefs['delivery_mode']
regular
>>> print global_prefs['delivery_status']
enabled
>>> print global_prefs['hide_address']
True
>>> print global_prefs['preferred_language']
en
>>> print global_prefs['receive_list_copy']
True
>>> print global_prefs['receive_own_postings']
True
```

Preferences can be set, but you have to call `save` to make your changes permanent.

Owners and Moderators

Owners and moderators are properties of the list object.

```
>>> test_one.owners
[]
>>> test_one.moderators
[]
```

Owners can be added via the `add_owner` method:

```
>>> test_one.add_owner('foo@example.com')
>>> for owner in test_one.owners:
...     print owner
foo@example.com
```

The owner of the list not automatically added as a member:

```
>>> test_one.members
[<Member "bill@example.com" on "test-one.example.com">]
```

Moderators can be added similarly:

```
>>> test_one.add_moderator('bar@example.com')
>>> for moderator in test_one.moderators:
...     print moderator
bar@example.com
```

Moderators are also not automatically added as members:

```
>>> test_one.members
[<Member "bill@example.com" on "test-one.example.com">]
```

Members and owners/moderators are separate entries in in the general members list:

```
>>> test_one.subscribe('bar@example.com')
<Member "bar@example.com" on "test-one.example.com">

>>> for member in client.members:
...     print '%s: %s' %(member, member.role)
<Member "foo@example.com" on "test-one.example.com">: owner
<Member "bar@example.com" on "test-one.example.com">: moderator
<Member "bar@example.com" on "test-one.example.com">: member
<Member "bill@example.com" on "test-one.example.com">: member
<Member "anna@example.com" on "test-two.example.com">: member
```

Both owners and moderators can be removed:

```
>>> test_one.remove_owner('foo@example.com')
>>> test_one.owners
[]
```

test_one.remove_moderator('bar@example.com') test_one.moderators []

Moderation

Message Moderation

```
>>> msg = """From: nomember@example.com
... To: test-one@example.com
... Subject: Something
... Message-ID: <moderated_01>
...
... Some text.
...
... """
>>> server = smtplib.LMTP('localhost', 8024)
>>> server.sendmail('nomember@example.com', 'test-one@example.com', msg)
{}
>>> server.quit()
(221, 'Bye')
>>> time.sleep(2)
```

Messages held for moderation can be listed on a per list basis.

```
>>> held = test_one.held
>>> print held[0]['subject']
Something
>>> print held[0]['reason']
```

```
>>> print held[0]['request_id']
1

>>> print test_one.defer_message(held[0]['request_id'])['status']
204

>>> len(test_one.held)
1

>>> print test_one.discard_message(held[0]['request_id'])['status']
204

>>> len(test_one.held)
0
```

Subscription Moderation

```
>>> patcher = patch('mailmanclient._client._Connection.call')
>>> mock_call = patcher.start()
>>> mock_call.return_value = [None, dict(entries=[
...     dict(address=u'ler@primus.org',
...          delivery_mode=u'regular',
...          display_name=u'Ler',
...          http_etag=u'...',
...          language=u'en',
...          password=u'password',
...          request_id=1,
...          type=u'subscription',
...          when=u'2005-08-01T07:49:23',
...     )
... ])]

>>> requests = test_one.requests
>>> print requests[0]['address']
ler@primus.org
>>> print requests[0]['delivery_mode']
regular
>>> print requests[0]['display_name']
Ler
>>> print requests[0]['language']
en
>>> print requests[0]['password']
password
>>> print requests[0]['request_id']
1
>>> print requests[0]['type']
subscription
>>> print requests[0]['request_date']
2005-08-01T07:49:23

>>> patcher.stop()
```

1.5.2 The Mailman Command Line Shell

This document describes the usage of the Mailman Command line shell, using which you can query a mailman installation with ease.

Firing and Exiting the Shell

You can start the mailman shell by executing the mmclient command, without any arguments:

```
$ ./mmclient
Mailman Command Line Interface
>>>
```

To exit the shell, use the `EOF` character, that is, `Ctrl + d`.

Displaying Mailman Objects

The shell can be used to display the mailman objects, using the show command.

For example:

```
>>> show users
>>> show domains
>>> show lists
```

Further, the CLI supports filtering of mailman objects based upon their attribute values or properties, using a *where* clause. For this, the CLI employs 3 filters, which are:

```
=         Equality
like      Case insensitive regular exp mathcing
in        Search inside a property which list
```

These filteres can be used in conjunction by using an `and` clause

Examples:

```
>>> show users where 'display_name' = 'Foo'
>>> show users where 'display_name' like '.*Foo*'
>>> show lists where 'foo@bar.com' in 'moderators'
>>> show lists where 'foo@bar.com' in 'moderators' and 'a@b.com' in 'owners'
```

The Shell Environment

The shell provides a facility to create variables that can be used to make the querying easier.

For using the shell, two commands, `set` and `unset` are used.

Example:

```
>>> set 'useremail' = 'foo@bar.com'
```

The variables can be used in the queries as follows:

```
>>> show lists where '$useremail' in 'moderators'
```

The `$username` will be replaced with the value of `useremail`

The environment can be disabled using the *disable environemt* command, that prevents the CLI from replacing the query terms with environment variables, or appending of the specialised variables.

The disabled environment can be enabled using the *enable env* command.:

```
>>> disable env
>>> enable env
```

The environment supports a set of special variables, which denote the names of the scopes available in mailman. They are domain, list and user.

The special environment variables are appended automatically with relevant commands

For example, if the environment variable domain is set to a domain name, then the ‘show list ‘ command automatically applies a domain = <set domain> filter to the result list.:

```
>>> set 'domain' = 'domain.org'
>>> show lists            //Shows lists under domain.org
>>> disable env
>>> show lists            //Shows all lists
>>> enable env
```

The value of stored variables can be viewed using the show_var command:

```
>>> show_var 'domain'
```

Create Objects

The Mailman objects can be created using the *create* command

The create command accepts the object properties and creates the object.

If the supplied arguments are invalid or insufficient, the list of arguments that are required are displayed.

The create command can be used as follows:

```
>>> create list where 'fqdn_listname' = 'list@domain.org'
>>> create domain where 'domain' = 'domain.org' and 'contact' = 'a@b.com'
>>> create user where 'email' = 'foo@bar.com' and 'password' = 'a' and 'name' = 'Foo'
```

Delete Objects

The Mailman objects can be deleted using the delete command. The delete command supports the same filters as those by the show command.

For example:

```
>>> delete domain where 'domain' like 'test_.*'
```

Subscription

The subscription commands include two commands, subscribe and unsubscribe users, which are respectively used to subscribe users to a list and unsubscribe users from a list. The commands allow applying the action on a single user or multiple users at a time.:

```
>>> subscribe users 'a@b.com' 'foo@bar.com' to 'list@domain.org'
>>> unsubscribe users 'a@b.com' 'foo@bar.com' from 'list@domain.org'
```

Update Preferences

Preferences can be updated using the shell for the following domains

- Globally
- Users

- Members
- Addresses

The actions are performed using the update command which can be used as follows:

```
>>> update preference '<preference_name>' to '<value>' globally
>>> update preference '<preference_name>' to '<value>' for member with 'email' = 'foo@bar.com'
    and 'list' = 'list@domain.org'
>>> update preference '<preference_name>' to '<value>' for user with 'email' = 'foo@bar.com'
>>> update preference '<preference_name>' to '<value>' for address with 'email' = 'foo@bar.com'
```

1.5.3 The Mailman Command Line Tools

Initialization

The CLI can be started by running mmclient [options|arguments]

If the mmclient is run without any arguments, the shell is started, else the specified action is performed. Use EOF to exit the shell.:

```
$> mmclient
Mailman Command Line Interface v1.0
>>>
Bye!

$> mmclient [options]
```

If you have non-default login credentials, specify them with the following options:

```
--host HOSTNAME       [defaults to http://127.0.0.1]
--port PORTNUMBER     [defaults to 8001]
--restuser USERNAME   [defaults to restadmin]
--restpass PASSWORD   [defaults to restpass]
```

Domains

This section describes the domain related functions that can be performed with the CLI.

Create a new domain

To create a new domain:

```
$> mmclient create domain testdomain.org
```

List Domains

To list the domains in the system:

```
$> mmclient show domain
http://domain.org
```

To obtain a detailed listing, use the *-v*/*–verbose* switch. The detailed listing of domains displays the domains as a table:

```
$> mmclient show domain -v
Base URL           Contact address        Mail host    URL host
http://domain.org  postmaster@domain.org  domain.org   domain.org
```

In addition, the long listing has a *no-header* switch that can be used to remove the header, making it more comfortable to pipe the output.:

```
$> mmclient show domain -v --no-header
http://domain.org  postmaster@domain.org  domain.org   domain.org
```

Delete a domain

To delete a domian :: $> mmclient delete domain domain.org

To supress the confirmation message:

```
$> mmclient delete domain domain.org --yes
```

To obtain a detailed description of a domain at one glance:

```
$> mmclient show domain domain.org
```

Mailing List

This section describes the mailing list related function that can be performed with the CLI.

Create a mailing list

To create a mailing list:

```
$> mmclient create list list@domain.org
```

Show Mailing lists

Show all mailing lists in the system:

```
$> mmclient show list
foo.domain.org
bar.example.org
```

To display the lists under the domain *domain.org*:

```
$> mmclient show list -d domain.org
foo.domain.org
```

Further, a switch -v/–verbose can also be used to print a detailed listing:

```
$> mmclient show list --verbose
ID               Name    Mail host    Display Name    FQDN
list.domain.org  list    domain.org   List            list@domain.org
```

Again, the *show list* supports a *–no-header* switch that removes the header line of the long listing:

```
$> mmclient show list --verbose --no-header
list.domain.org  list    domain.org   List            list@domain.org
```

Delete Mailing list

To delete a list:

```
$> mmclient delete list list@domain.org
```

To supress the confirmation message, do:

```
$> mmclient delete list list@domain.org --yes
```

To obtain a detailed description of a list at one glance:

```
$> mmclient show list list@domain.org
```

Manage list owners, members and moderators::

Adding and removing moderators can be performed using the CLI as follows

To get a list of members of a mailing list:

```
$> mmclient show-members list list@domain.org
```

The above command is equivalent to:

```
$> mmclient show user --list list@domain.com
```

The command also supports flags like:

```
--verbose Show a detailed listing
--no-header Hide header in detailed listing
```

Refer the *show user* command for more details

Add or remove list owners and moderators

To add moderators or owners:

```
$> mmclient add-moderator list list@domain.org --user a@b.com b@c.com
$> mmclient add-owner list list@domain.org --user a@b.com b@c.com
```

And to remove moderators or owners:

```
$> mmclient remove-moderator list list@domain.org --user a@b.com b@c.com
$> mmclient remove-owner list list@domain.org --user a@b.com b@c.com
```

The add and remove commands support the action on a list of users at once. Success or failure messages are provided upon completion or failure of an action. The messages can be supressed by using a quiet flag:

```
$> mmclient remove-moderator list list@domain.org --user a@b.com b@c.com --quiet
```

If the action fails for a user, the user is ignored and the action continues without stalling.

User

Craete User

To create a new user:

```
$> mmclient create user foo@bar.com --name "Foo" --password "password"
```

The *show user* command lists a single email ID of each user:

```
$> mmclient show user
foo@bar.com
```

To list users who are members of a list:

```
$> mmclient show user --list list@domain.org
foo@bar.com
```

The show command also supports a *–verbose* switch

```
$> mmclient show user --verbose
Display Name    Address       Created on                                                  User ID
Foo             foo2@bar.com  2014-05-30T00:52:52.564634  220337223817757552725201672981303248133
```

and a *–no-header* switch:

```
$> mmclient show user --verbose --no-header
Foo             foo2@bar.com  2014-05-30T00:52:52.564634  220337223817757552725201672981303248133
```

Delete User

To delete a user:

```
$> mmclient delete user foo@bar.com
```

To supress the confirmation message, do:

```
$> mmclient delete user foo@bar.com --yes
```

Describe user

To obtain a detailed description of a user at one glance:

```
$> mmclient show user foo@bar.com
```

Subscription and Unsubscription

Users can be subscribed to a mailing list by using the subscribe command:

```
$> mmclient subscribe user user1@bar.com user2@bar.com --list list@domain.org
user1@bar.com subscribed to list@domain.org
Failed to subscribe user2@bar.com : HTTP Error 409: Member already subscribed
```

Multiple users can be subscribed to the list at the same time.

Similarly, Users can be unsubscribed to a mailing list by using the unsubscribe command:

```
$> mmclient unsubscribe user user1@bar.com user2@bar.com --list list@domain.org
user1@bar.com unsubscribed from list@domain.org
user2@bar.com unsubscribed from list@domain.org
```

The feedback for the subscribe and unsubscribe actions can be supressed by using the –quiet flag:

```
$> mmclient subscribe user user1@bar.com user2@bar.com --list list@domain.org --quiet
$> mmclient unsubscribe user user1@bar.com user2@bar.com --list list@domain.org --quiet
```

The subscribe and unsubscribe actions continue even if one the users fail to subscribe/unsubscribe to a list. A relevant feedback message is provided if the –quiet flag is not enabled.

Preferences

Update Preference

Preferences for user,address,member or globally can be updated and retrieved by using the commands of preference scope.

To update the value of a preference of an user:

```
$> mmclient update preference user --email foo@bar.com [key] [value]
```

To update the value of a preference of a member:

```
$> mmclient update preference member --email foo@bar.com --list list@domain.org [key] [value]
```

To update the value of a preference of an address:

```
$> mmclient update preference address --email foo@bar.com [key] [value]
```

To update the value of a preference globally:

```
$> mmclient update preference global [key] [value]
```

View Setting

To view the current value of a preference, use the *mmclient show preference* command in the same way above, obviously, without the *value* argument

For eg, to view a setting of a member:

```
$> mmclient show preference member --email foo@bar.com --list list@domain.org [key]
```

Both the commands try to suggest the possible preference keys upon errors while typing the keys. The commands return *1* upon an invalid key.

Backup and Restore

The CLI tools can be used to create backups and restore the backups of the Mailman data. The backup currently supports the backup for SQLite mode.

Backup

The backup tools backs up the $var_dir specified by the Mailman configuration as a zip archive to a specified location:

```
$> mmclient backup ~/backup.zip
```

Restore

To restore the backup, specfiy the path to the backup file:

```
$> mmclient restore ~/backup.zip
```

Please remember to stop the mailman runner before performing the backup and restore operations.

The paths for backup and restore are read from the Mailman configuration file.

1.5.4 Writing a new command

There are two types of commands in the CLI project, either add a new command to the CLI tools, that gives a normal terminal command, or build a command for the Mailman custom shell.

Writing a new command for Command tools

To write a new command for the command line tools, the following steps are to be followed.

- Decide a command name and arguments expected
- Decide whether the command comes under a scope. Currently

the following scopes are supported

- users
- domains
- preferences
- lists

The new command can either be under any of these scopes or can exist independently without a scope.

- If the new command BAR is to be added to an existing scope `FOO`, the command would look like `FOO BAR`, where `FOO` is the action and BAR is the scope
- Add a new parser to the *action* parser as:

  ```
  new_cmd = action.add_parser('FOO')
  ```

  Add the scope as follows:

  ```
  scope = new_cmd.add_subparser(dest='scope')
  FOO_BAR = scope.add_parser('BAR')
  ```

- Add the arguments as follows:

  ```
  FOO_BAR.add_argument('-x','--xxxx',help='<help text>',[required=True/False])
  ```

- Add the action method in the `core/bar.py` file, to the Bar class. In effect, the action method would be `core.bar.Bar.action`. Note that the function name must be same as the action name

- If there is no specific scope, skip the scope parser and add the action the Misc class under `core/misc.py`. The action would be `core.misc.Misc.action`

Writing a new command for the Shell

To write a new shell command, first add a new parser for the command to the `client/parsers` directory.

Add the command method to the `client/shell.py`

Import the parser in the corresponding method and parse the user input, present in the `self.line` attribute to obtain the arguments. Perform the action.

1.5.5 Mailman Shell Parsers

This directory consists of the various parsers that have been built for use with the mailman shell. The shell parser has been built in such a way that each command has a different and independent parser, so that adding new commands can be done flexibly and easily, without worrying about the existing code.

The *show* command

The command to display the mailman objects

Tokens

```
SHOW   : 'show'
SCOPE  : '(user|domain|list)s?'
WHERE  : 'where'
OP     : '=|in|like'
AND    : 'and'
STRING : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
```

Grammar

```
S : SHOW SCOPE FILTER ;
FILTER : WHERE EXP | ;
EXP : STRING OP STRING CONJ ;
CONJ : AND EXP | ;
```

The *create* command

The commands to create mailman objects

Tokens

```
CREATE : 'create'
SCOPE  : 'user|domain|list'
WITH   : 'with'
AND    : 'and'
STRING : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
```

Grammar

```
S : CREATE SCOPE WITH EXP ;
EXP : STRING "=" STRING CONJ ;
CONJ : AND EXP | ;
```

The *delete* command

The commands to delete mailman objects

Tokens

```
DELETE : 'delete'
SCOPE  : '(user|domain|list)s?'
WHERE  : 'where'
OP     : '=|in|like'
AND    : 'and'
STRING : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
```

Grammar

```
S : DELETE SCOPE FILTER ;
FILTER : WHERE EXP | ;
EXP : STRING OP STRING CONJ ;
CONJ : AND EXP | ;
```

The *subscribe* Command

The commands to subscribe a list of users to a mailing lists

Tokens

```
SUBSCRIBE  : 'subscribe'
USER       : '(user)?'
TO         : 'to'
STRING : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
```

Grammar

```
S : SUBSCRIBE USER USERLIST TO STRING ;
USERLIST : STRING NEXT ;
NEXT : USERLIST | ;
```

The *unsubscribe* Command

The commands to unsubscribe a list of users from a mailing lists

Tokens

```
UNSUBSCRIBE : 'unsubscribe'
USER        : '(user)?'
FROM        : 'from'
STRING      : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
```

Grammar

```
S : UNSUBSCRIBE USER USERLIST TO STRING ;
USERLIST : STRING NEXT ;
NEXT : USERLIST | ;
```

The *unset* command

Command to unset a shell variable

Tokens

```
UNSET : 'unset'
STRING      : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
```

Grammar

```
S : UNSET STRING ;
```

The *set* Command

Command to set a shell variable

Tokens

```
SET : 'set'
STRING      : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
```

Grammar

```
S : SET STRING "=" STRING ;
```

The *update* command

Command to update a preference

Tokens

```
UPDATE      : 'update'
PREFERENCE  : 'preference'
STRING      : '`([a-zA-Z0-9_@\.\*\-\$ ]*)`'
TO          : 'to'
WITH        : 'with'
AND         : 'and'
FOR         : 'for'
GLOBALLY    : 'globally'
DOMAIN      : 'user|address|member'
```

Grammar

```
S : UPDATE PREFERENCE STRING TO STRING E ;
E : GLOBALLY | FOR DOMAIN WITH EXP ;
EXP : STRING "=" STRING CONJ ;
CONJ : AND EXP | ;
```

1.5.6 NEWS for mailman.client

0.1 (201X-XX-XX)

- Initial release.