Mailman Client Documentation

Release 3.2.1

Mailman Coders

Jul 14, 2018

Table of Contents

1 Requirements **3**

2 Documentation **5**

3 Project details **7**

4 Acknowledgements **9**

4.1 NEWS for mailmanclient 9

4.2 Example Usage 10

4.3 API Reference 32

4.4 Developing MailmanClient 32

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

CHAPTER 1

Requirements

`mailmanclient` requires Python 2.7 or newer.

CHAPTER 2

Documentation

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

http://mailmanclient.readthedocs.org/en/latest/

CHAPTER 3

Project details

The project home page is:

https://gitlab.com/mailman/mailmanclient

You should report bugs at:

https://gitlab.com/mailman/mailmanclient/issues

You can download the latest version of the package either from the Cheese Shop:

http://pypi.python.org/pypi/mailmanclient

or from the GitLab page above. Of course you can also just install it with `pip` from the command line:

```
$ pip install mailmanclient
```

You can grab the latest development copy of the code using Git, from the Gitlab home page above. If you have Git installed, you can grab your own branch of the code like this:

```
$ git clone https://gitlab.com/mailman/mailmanclient.git
```

You may contact the developers via mailman-developers@python.org

CHAPTER 4

Acknowledgements

Many thanks to Florian Fuchs for his contribution of an initial REST client. Also thanks to all the contributors of Mailman Client who have contributed code, raised issues or devoted their time in any capacity!

4.1 NEWS for mailmanclient

4.1.1 3.2.1 (201X-XX-XX)

- Add support for Python 3.7

4.1.2 3.2.0 (2018-07-10)

Changes

- **Add '.pc' (patch directory) to list of ignored patterns when building the** documentation with Sphinx.
- *Mailinglist.add_owner* and *Mailinglist.add_moderator* now accept an additional *display_name* argument that allows associating display names with these memberships.
- **Add a new API `Client.find_lists` which allows filtering mailing lists** related to a subscriber. It optionally allows a role, which filters the lists that the address is subscribed to with that role.

Backwards Incompatible Changes

- *MailingList.owners* and *MailingList.moderators* now returns a list of *Member* objects instead of a list of emails.

Backwards Incompatible Changes

- *Domain.owners* now returns a list of *User* objects instead of just a dictionary of JSON response. (!63)

4.1.3 3.1.1 (2017-10-07)

- Python3 compatibility is fixed, mailmanclient is now compatible through Python2.7 - Python3.6
- Internal source code is now split into several class-specific modules as compared to previously a single giant _client module.
- All the RestObjects, like MailingList, are now exposed from the top level import.
- Old *mailmanclient._client* module is added back for compatibility with versions of Postorius that use some internal APIs.

4.1.4 3.1 (2017-05-25)

- Bug fixes.
- Align with Mailman 3.1 Core REST API.
- Python3 compatibility is broken because of a urllib bug.

4.1.5 1.0.1 (2015-11-14)

- Bugfix release.

4.1.6 1.0.0 (2015-04-17)

- Port to Python 3.4.
- Run test suite with *tox*.
- Use vcrpy for HTTP testing.
- Add list archiver access.
- Add subscription moderation

4.1.7 1.0.0a1 (2014-03-15)

- Initial release.

4.2 Example Usage

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:9001/3.1', 'restadmin', 'restpass')
```

Note: Please note that port '9001' is used above, since mailman's test server runs on port *9001*. In production Mailman's REST API usually listens on port *8001*.

We can retrieve basic information about the server.

```
>>> dump(client.system)
api_version: 3.1
http_etag: "..."
mailman_version: GNU Mailman ... (...)
python_version: ...
self_link: http://localhost:9001/3.1/system/versions
```

To start with, there are no known mailing lists.

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

4.2.1 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')
>>> print(example_dot_com.description)
None
>>> print(example_dot_com.mail_host)
example.com
>>> print(example_dot_com.alias_domain)
None
```

A domain can have an alias_domain attribute to help with some unusual Postfix configurations.

```
>>> example_dot_edu = client.create_domain('example.edu',
...                                        alias_domain='x.example.edu')
>>> print(example_dot_edu.mail_host)
example.edu
>>> print(example_dot_edu.alias_domain)
x.example.edu
```

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

```
>>> example = client.get_domain('example.com')
>>> print(example.mail_host)
example.com
```

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

```
>>> example_net = client.create_domain('example.net')
>>> example_org = client.create_domain('example.org')
>>> print(example_org.mail_host)
example.org
>>> for domain in client.domains:
...     print(domain.mail_host)
example.com
```

(continues on next page)

(continued from previous page)

```
example.edu
example.net
example.org
```

Also, domain can be deleted.

```
>>> example_org.delete()
>>> for domain in client.domains:
...     print(domain.mail_host)
example.com
example.edu
example.net
```

4.2.2 Mailing lists

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

```
>>> test_one = example.create_list('test-1')
>>> print(test_one.fqdn_listname)
test-1@example.com
>>> print(test_one.mail_host)
example.com
>>> print(test_one.list_name)
test-1
>>> print(test_one.display_name)
Test-1
```

You can create a mailing list with a specific list style.

```
>>> test_two = example.create_list('test-announce', style_name='legacy-announce')
>>> print(test_two.fqdn_listname)
test-announce@example.com
```

You can retrieve a list of known mailing list styles along with the default one.

```
>>> styles = client.styles
>>> from operator import itemgetter
>>> for style in sorted(styles['styles'], key=itemgetter('name')):
...     print('{0}: {1}'.format(style['name'], style['description']))
legacy-announce: Announce only mailing list style.
legacy-default: Ordinary discussion mailing list style.
>>> print(styles['default'])
legacy-default
```

You can also retrieve the mailing list after the fact.

```
>>> my_list = client.get_list('test-1@example.com')
>>> print(my_list.fqdn_listname)
test-1@example.com
```

And you can print all the known mailing lists.

```
>>> print(example.create_list('test-2').fqdn_listname)
test-2@example.com
>>> domain = client.get_domain('example.net')
```

(continues on next page)

(continued from previous page)

```
>>> print(domain.create_list('test-3').fqdn_listname)
test-3@example.net
>>> print(example.create_list('test-3').fqdn_listname)
test-3@example.com

>>> for mlist in client.lists:
...     print(mlist.fqdn_listname)
test-1@example.com
test-2@example.com
test-3@example.com
test-3@example.net
test-announce@example.com
```

You can also select advertised lists only.

```
>>> my_list.settings['advertised'] = False
>>> my_list.settings.save()
>>> for mlist in client.get_lists(advertised=True):
...     print(mlist.fqdn_listname)
test-2@example.com
test-3@example.com
test-3@example.net
test-announce@example.com
```

List results can be retrieved as pages:

```
>>> page = client.get_list_page(count=2, page=1)
>>> page.nr
1
>>> len(page)
2
>>> page.total_size
5
>>> for m_list in page:
...     print(m_list.fqdn_listname)
test-1@example.com
test-2@example.com
>>> page = page.next
>>> page.nr
2
>>> for m_list in page:
...     print(m_list.fqdn_listname)
test-3@example.com
test-3@example.net
```

Pages can also use the advertised filter:

```
>>> page = client.get_list_page(count=2, page=1, advertised=True)
>>> for m_list in page:
...     print(m_list.fqdn_listname)
test-2@example.com
test-3@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.fqdn_listname)
```

(continues on next page)

(continued from previous page)

```
test-1@example.com
test-2@example.com
test-3@example.com
test-announce@example.com
```

It is also possible to display only advertised lists when using the domain.

```
>>> for mlist in example.get_lists(advertised=True):
...     print(mlist.fqdn_listname)
test-2@example.com
test-3@example.com
test-announce@example.com
>>> for mlist in example.get_list_page(count=2, page=1, advertised=True):
...     print(mlist.fqdn_listname)
test-2@example.com
test-3@example.com
```

You can use a list instance to delete the list.

```
>>> test_three = client.get_list('test-3@example.net')
>>> test_three.delete()
```

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

```
>>> client.delete_list('test-3@example.com')
```

```
>>> for mlist in client.lists:
...     print(mlist.fqdn_listname)
test-1@example.com
test-2@example.com
test-announce@example.com
```

4.2.3 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-2@example.com')
>>> print(test_two.settings['subscription_policy'])
confirm
```

Email addresses need to be verified first, so if we try to subscribe a user, we get a response with a token:

```
>>> data = test_one.subscribe('unverified@example.com', 'Unverified')
>>> data['token'] is not None
True
```

(continues on next page)

(continued from previous page)

```
>>> print(data['token_owner'])
subscriber
```

If we know the email address to be valid, we can set the `pre_verified` flag. However, the list's subscription policy is "confirm", so if we try to subscribe a user, we will also get a token back:

```
>>> data = test_one.subscribe('unconfirmed@example.com',
...                           'Unconfirmed',
...                           pre_verified=True)
>>> data['token'] is not None
True
>>> print(data['token_owner'])
subscriber
```

If we know the user originated the subscription (for example if she or he has been authenticated elsewhere), we can set the `pre_confirmed` flag.

The `pre_approved` flag is used for lists that require moderator approval and should only be used if the subscription is initiated by a moderator or admin.

```
>>> print(test_one.subscribe('anna@example.com', 'Anna',
...                          pre_verified=True,
...                          pre_confirmed=True))
Member "anna@example.com" on "test-1.example.com"
```

```
>>> print(test_one.subscribe('bill@example.com', 'Bill',
...                          pre_verified=True,
...                          pre_confirmed=True))
Member "bill@example.com" on "test-1.example.com"
```

```
>>> print(test_two.subscribe('anna@example.com',
...                          pre_verified=True,
...                          pre_confirmed=True))
Member "anna@example.com" on "test-2.example.com"
```

```
>>> print(test_two.subscribe('cris@example.com', 'Cris',
...                          pre_verified=True,
...                          pre_confirmed=True))
Member "cris@example.com" on "test-2.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-1.example.com"
Member "bill@example.com" on "test-1.example.com"
Member "anna@example.com" on "test-2.example.com"
Member "cris@example.com" on "test-2.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-1.example.com"
Member "bill@example.com" on "test-1.example.com"
```

Membership may have a name associated, this depends on whether the member `Address` or `User` has a `display_name` attribute.

```
>>> for member in test_one.members:
...     print(member.display_name)
Anna
Bill
```

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
>>> page.total_size
4
>>> for member in page:
...     print(member)
Member "anna@example.com" on "test-1.example.com"
Member "bill@example.com" on "test-1.example.com"
```

```
>>> page = page.next
>>> page.nr
2
>>> for member in page:
...     print(member)
Member "anna@example.com" on "test-2.example.com"
Member "cris@example.com" on "test-2.example.com"
```

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

We can get a single membership too.

```
>>> cris_test_two = test_two.get_member('cris@example.com')
>>> print(cris_test_two)
Member "cris@example.com" on "test-2.example.com"
>>> print(cris_test_two.role)
member
>>> print(cris_test_two.display_name)
Cris
```

A membership can also be retrieved without instantiating the list object first:

```
>>> print(client.get_member('test-2@example.com', 'cris@example.com'))
Member "cris@example.com" on "test-2.example.com"
```

A membership has preferences.

```
>>> prefs = cris_test_two.preferences
>>> print(prefs['delivery_mode'])
None
>>> 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_u = cris_test_two.user
>>> print(cris_u.display_name, cris_u.user_id)
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-2@example.com
```

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

```
>>> import time
>>> time.sleep(2)
>>> test_one.unsubscribe('anna@example.com')
>>> for member in client.members:
...     print(member)
Member "bill@example.com" on "test-1.example.com"
Member "anna@example.com" on "test-2.example.com"
Member "cris@example.com" on "test-2.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-1.example.com"
Member "anna@example.com" on "test-2.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-1@example.com
```

4.2.4 Non-Members

When someone attempts to post to a list but is not a member, then they are listed as a "non-member" of that list so that a moderator can choose how to handle their messages going forward. In some cases, one might wish to accept or reject their future messages automatically. Just like with regular members, they are given a unique id.

The list starts out with no nonmembers.

```
>>> test_one.nonmembers
[]
```

When someone tries to send a message to the list and they are not a subscriber, they get added to the nonmember list.

4.2.5 Users

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

```
>>> for user in client.users:
...     print(user.display_name)
Unverified
Unconfirmed
Anna
Bill
Cris
```

The list of users can also be paginated:

```
>>> page = client.get_user_page(count=4, page=1)
>>> page.nr
1
>>> page.total_size
5
```

```
>>> for user in page:
...     print(user.display_name)
Unverified
Unconfirmed
Anna
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.display_name)
Cris
```

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

```
>>> for user in page:
...     print(user.display_name)
Unverified
Unconfirmed
Anna
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
...
```

Multiple addresses can be assigned to a user record:

```
>>> print(cris.add_address('cris.person@example.org'))
cris.person@example.org
>>> print(client.get_address('cris.person@example.org'))
cris.person@example.org
```

```
>>> for address in cris.addresses:
...     print(address)
cris.person@example.org
cris@example.com
```

Trying to add an existing address will raise an error:

```
>>> dana = client.create_user(email='dana@example.org',
...                           password='somepass',
...                           display_name='Dana')
>>> print(dana.display_name)
Dana
>>> cris.add_address('dana@example.org')
Traceback (most recent call last):
...
HTTPError: HTTP Error 400: Address already exists
```

This can be overridden by using the `absorb_existing` flag:

```
>>> print(cris.add_address('dana@example.org', absorb_existing=True))
dana@example.org
```

The user Chris will then be merged with Dana, acquiring all its subscriptions and preferences. In case of conflict, Chris' original preferences will prevail.

```
>>> for address in cris.addresses:
...     print(address)
cris.person@example.org
cris@example.com
dana@example.org
```

4.2.6 Addresses

Addresses can be accessed directly:

```
>>> address = client.get_address('dana@example.org')
>>> print(address)
dana@example.org
>>> print(address.display_name)
Dana
```

The address has not been verified:

```
>>> print(address.verified)
False
```

But that can be done via the address object:

```
>>> address.verify()
>>> print(address.verified)
True
```

It can also be unverified:

```
>>> address.unverify()
>>> print(address.verified)
False
```

Addresses can be deleted by calling their `delete()` method or by removing them from their user's `addresses` list:

```
>>> cris.addresses.remove('dana@example.org')
>>> for address in cris.addresses:
...     print(address)
cris.person@example.org
cris@example.com
```

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

```
>>> ler = client.create_user(email='ler@primus.org',
...                          password='somepass',
...                          display_name='Ler')
>>> print(ler.display_name)
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-1.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-1.example.com
```

4.2.7 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)
57
```

```
>>> for attr in sorted(settings):
...     print(attr + ': ' + str(settings[attr]))
acceptable_aliases: []
...
volume: 1
```

```
>>> print(settings['display_name'])
Test-1
```

We can access all valid list settings as attributes.

```
>>> print(settings['fqdn_listname'])
test-1@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
```

4.2.8 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.

```
>>> prefs = test_two.get_member('anna@example.com').preferences
>>> prefs['delivery_status'] = 'by_user'
>>> prefs.save()
>>> prefs = test_two.get_member('anna@example.com').preferences
>>> print(prefs['delivery_status'])
by_user
```

4.2.9 Pipelines and Chains

The available pipelines and chains can also be retrieved:

```
>>> pipelines = client.pipelines['pipelines']
>>> for pipeline in pipelines:
...     print(pipeline)
default-owner-pipeline
default-posting-pipeline
virgin
>>> chains = client.chains['chains']
>>> for chain in chains:
...     print(chain)
accept
default-owner-chain
default-posting-chain
discard
dmarc
header-match
hold
moderation
reject
```

4.2.10 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 and they can have an optional `display_name` associated like other `members`:

```
>>> test_one.add_owner('foo@example.com', display_name='Foo')
>>> for owner in test_one.owners:
...     print(owner.email)
foo@example.com
```

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

```
>>> for m in test_one.members:
...     print(m)
Member "bill@example.com" on "test-1.example.com"
```

Moderators can be added similarly:

```
>>> test_one.add_moderator('bar@example.com', display_name='Bar')
>>> for moderator in test_one.moderators:
...     print(moderator.email)
bar@example.com
```

Moderators are also not automatically added as members:

```
>>> for m in test_one.members:
...     print(m)
Member "bill@example.com" on "test-1.example.com"
```

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

```
>>> print(test_one.subscribe('bar@example.com', 'Bar',
...                          pre_verified=True,
...                          pre_confirmed=True))
Member "bar@example.com" on "test-1.example.com"
```

```
>>> for member in client.members:
...     print('%s: %s' % (member, member.role))
Member "foo@example.com" on "test-1.example.com": owner
Member "bar@example.com" on "test-1.example.com": moderator
Member "bar@example.com" on "test-1.example.com": member
Member "bill@example.com" on "test-1.example.com": member
Member "anna@example.com" on "test-2.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 []

4.2.11 Moderation

Subscription Moderation

Subscription requests can be accessed through the list object's *request* property. So let's create a non-open list first.

```
>>> confirm_first = example_dot_com.create_list('confirm-first')
>>> settings = confirm_first.settings
>>> settings['subscription_policy'] = 'moderate'
>>> settings.save()
```

```
>>> confirm_first = client.get_list('confirm-first.example.com')
>>> print(confirm_first.settings['subscription_policy'])
moderate
```

Initially there are no requests, so let's to subscribe someone to the list. We'll get a token back.

```
>>> confirm_first.requests
[]
>>> data = confirm_first.subscribe('groucho@example.com',
...                                pre_verified=True,
...                                pre_confirmed=True)
>>> print(data['token_owner'])
moderator
```

Now the request shows up in the list of requests:

```
>>> import time; time.sleep(5)
>>> len(confirm_first.requests)
1
```

```
>>> request_1 = confirm_first.requests[0]
>>> print(request_1['email'])
groucho@example.com
>>> print (request_1['token'] is not None)
True
>>> print(request_1['token_owner'])
moderator
>>> print(request_1['request_date'] is not None)
True
>>> print(request_1['list_id'])
confirm-first.example.com
```

Subscription requests can be accepted, deferred, rejected or discarded using the request token.

```
>>> data = confirm_first.subscribe('harpo@example.com',
...                                pre_verified=True,
...                                pre_confirmed=True)
>>> data = confirm_first.subscribe('zeppo@example.com',
...                                pre_verified=True,
...                                pre_confirmed=True)
```

```
>>> len(confirm_first.requests)
3
```

Let's accept Groucho:

```
>>> response = confirm_first.moderate_request(request_1['token'], 'accept')
>>> len(confirm_first.requests)
2
```

```
>>> request_2 = confirm_first.requests[0]
>>> print(request_2['email'])
harpo@example.com
```

```
>>> request_3 = confirm_first.requests[1]
>>> print(request_3['email'])
zeppo@example.com
```

Let's reject Harpo:

```
>>> response = confirm_first.moderate_request(request_2['token'], 'reject')
>>> len(confirm_first.requests)
1
```

Let's discard Zeppo's request:

```
>>> response = confirm_first.moderate_request(request_3['token'], 'discard')
>>> len(confirm_first.requests)
0
```

Message Moderation

By injecting a message by a non-member into the incoming queue, we can simulate a message being held for moderator approval.

```
>>> msg = """From: nomember@example.com
... To: test-1@example.com
... Subject: Something
... Message-ID: <moderated_01>
...
... Some text.
...
... """
>>> inq = client.queues['in']
>>> inq.inject('test-1.example.com', msg)
```

Now wait until the message has been processed.

```
>>> while True:
...     if len(inq.files) == 0:
...         break
...     time.sleep(0.1)
```

It might take a few moments for the message to show up in the moderation queue.

```
>>> while True:
...     all_held = test_one.held
...     if len(all_held) > 0:
...         break
...     time.sleep(0.1)
```

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

```
>>> print(all_held[0].request_id)
1
```

A held message can be retrieved by ID, and have attributes:

```
>>> heldmsg = test_one.get_held_message(1)
>>> print(heldmsg.subject)
Something
>>> print(heldmsg.reason)
The message is not from a list member
>>> print(heldmsg.sender)
nomember@example.com
>>> 'Message-ID: <moderated_01>' in heldmsg.msg
True
```

A moderation action can be taken on them using the list methods or the held message's methods.

```
>>> print(test_one.defer_message(heldmsg.request_id)['status'])
204
```

```
>>> len(test_one.held)
1
```

```
>>> print(heldmsg.discard()['status'])
204
```

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

Member moderation

Each member or non-member can have a specific moderation action. It is set using the 'moderation_action' property:

```
>>> bill_member = test_one.get_member('bill@example.com')
>>> print(bill_member.moderation_action)
None
>>> bill_member.moderation_action = 'hold'
>>> bill_member.save()
>>> print(test_one.get_member('bill@example.com').moderation_action)
hold
```

Banning addresses

A ban list is a list of email addresses that are not allowed to subscribe to a mailing-list. There are two types of ban lists: each mailing-list has its ban list, and there is a site-wide list. Addresses on the site-wide list are prevented from subscribing to every mailing-list on the server.

To view the site-wide ban list, use the *bans* property:

```
>>> list(client.bans)
[]
```

You can use the *add* method on the ban list to ban an email address:

```
>>> banned_anna = client.bans.add('anna@example.com')
>>> print(banned_anna)
anna@example.com
>>> 'anna@example.com' in client.bans
True
>>> print(client.bans.add('bill@example.com'))
bill@example.com
>>> for addr in list(client.bans):
...     print(addr)
anna@example.com
bill@example.com
```

The list of banned addresses can be paginated using the `get_bans_page()` method:

```
>>> for addr in list(client.get_bans_page(count=1, page=1)):
...     print(addr)
anna@example.com
>>> for addr in list(client.get_bans_page(count=1, page=2)):
...     print(addr)
bill@example.com
```

You can use the `delete()` method on a banned address to unban it, or the `remove()` method on the ban list:

```
>>> banned_anna.delete()
>>> 'anna@example.com' in client.bans
False
>>> for addr in list(client.bans):
...     print(addr)
bill@example.com
>>> client.bans.remove('bill@example.com')
>>> 'bill@example.com' in client.bans
False
>>> print(list(client.bans))
[]
```

The mailing-list-specific ban lists work in the same way:

```
>>> print(list(test_one.bans))
[]
>>> banned_anna = test_one.bans.add('anna@example.com')
>>> 'anna@example.com' in test_one.bans
True
>>> print(test_one.bans.add('bill@example.com'))
bill@example.com
>>> for addr in list(test_one.bans):
...     print(addr)
anna@example.com
bill@example.com
>>> for addr in list(test_one.get_bans_page(count=1, page=1)):
...     print(addr)
anna@example.com
>>> for addr in list(test_one.get_bans_page(count=1, page=2)):
...     print(addr)
bill@example.com
>>> banned_anna.delete()
>>> 'anna@example.com' in test_one.bans
False
>>> test_one.bans.remove('bill@example.com')
>>> print(list(test_one.bans))
[]
```

4.2.12 Archivers

Each list object has an `archivers` attribute.

```
>>> archivers = test_one.archivers
>>> print(archivers)
Archivers on test-1.example.com
```

The activation status of each available archiver can be accessed like a key in a dictionary.

```
>>> archivers = test_one.archivers
>>> for archiver in sorted(archivers.keys()):
...     print('{0}: {1}'.format(archiver, archivers[archiver]))
mail-archive: True
mhonarc: True
prototype: True
```

```
>>> archivers['mail-archive']
True
>>> archivers['mhonarc']
True
```

They can also be set like items in dictionary.

```
>>> archivers['mail-archive'] = False
>>> archivers['mhonarc'] = False
```

So if we get a new `archivers` object from the API (by accessing the list's archiver attribute again), we can see that the archiver stati have now been set.

```
>>> archivers = test_one.archivers
>>> archivers['mail-archive']
False
>>> archivers['mhonarc']
False
```

4.2.13 Header matches

Header matches are filtering rules that apply to messages sent to a mailing list. They match a header to a pattern using a regular expression, and matching patterns can trigger specific moderation actions. They are accessible via the mailing list's `header_matches` attribute, which behaves like a list.

```
>>> header_matches = test_one.header_matches
>>> print(header_matches)
Header matches for "test-1.example.com"
>>> len(header_matches)
0
```

Header matches can be added using the `add()` method. The arguments are:

- the header to consider (`str`). Il will be lower-cased.
- the regular expression to use for filtering (`str`)
- the action to take when the header matches the pattern. This can be `'accept'`, `'discard'`, `'reject'`, or `'hold'`.

```
>>> print(header_matches.add('Subject', '^test: ', 'discard'))
Header match on "subject"
>>> print(header_matches)
Header matches for "test-1.example.com"
>>> len(header_matches)
1
>>> for hm in list(header_matches):
...     print(hm)
Header match on "subject"
```

You can delete a header match by deleting it from the `header_matches` collection.

```
>>> del header_matches[0]
>>> len(header_matches)
0
```

You can also delete a header match using its `delete()` method, but be aware that the collection will not automatically be updated. Get a new collection from the list's `header_matches` attribute to see the change.

```
>>> print(header_matches.add('Subject', '^test: ', 'discard'))
Header match on "subject"
>>> header_matches[0].delete()
>>> len(header_matches) # not automatically updated
1
>>> len(test_one.header_matches)
0
```

4.2.14 Configuration

Mailman Core exposes all its configuration through REST API. All these configuration options are read-only.

```
>>> cfg = client.configuration
>>> for key in sorted(cfg):
...     print(cfg[key].name)
antispam
archiver.mail_archive
archiver.master
archiver.mhonarc
archiver.prototype
bounces
database
devmode
digests
dmarc
language.ar
language.ast
language.ca
language.cs
language.da
language.de
language.el
language.en
language.es
language.et
language.eu
language.fi
language.fr
language.gl
language.he
language.hr
language.hu
language.ia
language.it
language.ja
language.ko
language.lt
language.nl
language.no
language.pl
language.pt
language.pt_BR
language.ro
```

(continues on next page)

(continued from previous page)

```
language.ru
language.sk
language.sl
language.sr
language.sv
language.tr
language.uk
language.vi
language.zh_CN
language.zh_TW
logging.archiver
logging.bounce
logging.config
logging.database
logging.debug
logging.error
logging.fromusenet
logging.http
logging.locks
logging.mischief
logging.plugins
logging.root
logging.runner
logging.smtp
logging.subscribe
logging.vette
mailman
mta
nntp
passwords
paths.dev
paths.fhs
paths.here
paths.local
plugin.master
runner.archive
runner.bad
runner.bounces
runner.command
runner.digest
runner.in
runner.lmtp
runner.nntp
runner.out
runner.pipeline
runner.rest
runner.retry
runner.shunt
runner.virgin
shell
styles
webservice
```

Each configuration object is a dictionary and you can iterate over them:

```
>>> for key in sorted(cfg['mailman']):
...     print('{} : {}'.format(key, cfg['mailman'][key]))
```

(continues on next page)

(continued from previous page)

```
cache_life : 7d
default_language : en
email_commands_max_lines : 10
filtered_messages_are_preservable : no
html_to_plain_text_command : /usr/bin/lynx -dump $filename
layout : here
listname_chars : [-_.0-9a-z]
noreply_address : noreply
pending_request_life : 3d
post_hook :
pre_hook :
self_link : http://localhost:9001/3.1/system/configuration/mailman
sender_headers : from from_ reply-to sender
site_owner : changeme@example.com
```

4.3 API Reference

4.4 Developing MailmanClient

4.4.1 Running Tests

The test suite is run with the tox tool, which allows it to be run against multiple versions of Python. There are two modes to the test suite:

- *Record mode* which is used to record the HTTP traffic against a live Mailman 3 REST server.
- *Replay mode* which allows you to run the test suite off-line, without running the REST server.

Whenever you add tests for other parts of the REST API, you need to run the suite once in record mode to generate the YAML file of HTTP requests and responses.

Then you can run the test suite in replay mode as often as you want, and Mailman 3 needn't even be installed on your system.

Since this branch ships with a recording file, you don't need to run in record mode to start with.

4.4.2 Replay mode

To run the test suite in replay mode (the default), just run the following:

```
$ tox
```

This will attempt to run the test suite against Python 2.7, 3.4, 3.5 and 3.6 or whatever combination of those that are available on your system.

4.4.3 Record mode

Start by branching the Mailman 3 code base, then you should install it into a virtual environment. The easiest way to do this is with *tox*:

```
$ tox --notest -r
```

Now, use the virtual environment that *tox* creates to create a template *var* directory in the current directory:

```
$ .tox/py34/bin/mailman info
```

Now you need to modify the `var/etc/mailman.cfg` configuration file, so that it contains the following:

```
[devmode]
enabled: yes
testing: yes
recipient: you@yourdomain.com

[mta]
smtp_port: 9025
lmtp_port: 9024
incoming: mailman.testing.mta.FakeMTA

[webservice]
port: 9001

[archiver.mhonarc]
enable: yes

[archiver.mail_archive]
enable: yes

[archiver.prototype]
enable: yes
```

Now you can start Mailman 3:

```
$ .tox/py34/bin/mailman start
```

Back in your `mailmanclient` branch, run the test suite in record mode:

```
$ tox -e record
```

You should now have an updated recording file (`tape.yaml`).

If you find you need to re-run the test suite, you *must* first stop the Mailman REST server, and then delete the `mailman.db` file, since it contains state that will mess up the `mailmanclient` test suite:

```
$ cd <mailman3-branch>
$ .tox/py34/bin/mailman stop
$ rm -f var/data/mailman.db
$ .tox/py34/bin/mailman start

$ cd <mailmanclient-branch>
$ tox -e record
```

Once you're done recording the HTTP traffic, you can stop the Mailman 3 server and you won't need it again. It's a good idea to commit the `tape.yaml` changes for other users of your branch.