Modoboa Documentation

Release 1.6.2

Antoine Nguyen

December 13, 2016

Contents

CHAPTER 1

Overview

Modoboa is a mail hosting and management platform including a modern and simplified Web User Interface designed to work with Postfix and Dovecot.

It is extensible by nature and comes with a lot of additional extensions:

CHAPTER 2

Table of contents

2.1 Installation

2.1.1 Recommended way

If you start from scratch and want to deploy a complete mail server, you will love the modoboa installer! It is the easiest and the quickest way to setup a fully functional server (modoboa, postfix, dovecot, amavis and more) on one machine.

Warning: For now, only Debian and CentOS based Linux distributions are supported. We do our best to improve compatibility but if you use another Linux or a UNIX system, you will have to install Modoboa *manually*.

To use it, just run the following commands in your terminal:

```
> git clone https://github.com/modoboa/modoboa-installer
> cd modoboa-installer
> sudo ./run.py <mail server hostname>
```

Wait a few minutes and you're done o/

2.1.2 Manual installation

For those who need a manual installation or who just want to setup a specific part, here are the steps you must follow:

Modoboa

This section describes the installation of the web interface (a Django project).

Prepare the system

First of all, we recommand the following context:

- Use a dedicated system user
- Use a virtualenv to install the application because it will isolate it (and its dependencies) from the rest of your system

The following example illustrates how to realize this (Debian like system):

```
> sudo apt-get install python-virtualenv python-pip
> sudo useradd modoboa
> sudo -i modoboa
> virtualenv env
(env)> source env/bin/activate
(env)> pip install -U pip
```

FIXME: dépendances système pour compilation

Then, install Modoboa:

```
(env)> pip install modoboa
```

Database

Warning: This documentation does not cover the installation of a database server but only the setup of a functional database that Modoboa will use.

Thanks to Django, Modoboa is compatible with the following databases:

- PostgreSQL
- MySQL / MariaDB
- SQLite

Since the last one does not require particular actions, only the first two ones are described.

PostgreSQL Install the corresponding Python binding:

```
(env)> pip install psycopg2
```

Then, create a user and a database:

```
> sudo -i postgres
>
```

MySQL / MariaDB Install the corresponding Python binding:

```
(env)> pip install MySQL-Python
```

Then, create a user and a database:

```
> mysqladmin -u root -p create modoboa
```

Deploy an instance

`modoboa-admin.py`, a command line tool, lets you deploy a *ready-to-use* Modoboa site using only one instruction:

```
(env)> modoboa-admin.py deploy instance --collectstatic \
         --domain <hostname of your server> --dburl default:database-url
```

Note: You can install additional extensions during the deploy process. To do so, use the `--extensions` option which accepts a list of names as argument (`--extensions ext1 ext2 ...`). If you want to install all extensions, just use the `all` keyword like this `--extensions all`.

If you choose to install extensions one at a time, you will have to add their names in settings.py to `MODOBOA_APPS`. Also ensure that you have the line `from modoboa_amavis.settings import *` at the end of this file.

The list of available extensions can be found on the index page. Instructions to install them are available on each extensions page.

Note: You can specify more than one database connection using the `--dburl` option. Multiple connections are differentiated by a prefix.

The primary connection must use the `default:` prefix (as shown in the example above). For the amavis extension, use the `amavis:` prefix. For example: `--dburl default:<database url> amavis:<database url>`.

A database url should meet the following syntax `<mysql|postgres>://[user:pass@][host:port]/dbname` **OR** `sqlite:////full/path/to/your/database/file.sqlite`.

The command will ask you a few questions, answer them and you're done.

If you need a **silent installation** (e.g. if you're using Salt-Stack, Ansible or whatever), it's possible to supply the database credentials as commandline arguments.

You can consult the complete option list by running the following command:

```
$ modoboa-admin.py help deploy
```

Cron jobs

A few recurring jobs must be configured to make Modoboa works as expected.

Create a new file, for example `/etc/cron.d/modoboa` and put the following content inside:

```
#
# Modoboa specific cron jobs
#
PYTHON=<PATH TO PYTHON BINARY>
INSTANCE=<PATH TO MODOBOA INSTANCE>

# Operations on mailboxes
*       *       *       *       *       vmail   $PYTHON $INSTANCE/manage.py handle_mailbox_operations

# Sessions table cleanup
0       0       *       *       *       root    $PYTHON $INSTANCE/manage.py clearsessions

# Logs table cleanup
0       0       *       *       *       root    $PYTHON $INSTANCE/manage.py cleanlogs

# Logs parsing
*/5     *       *       *       *       root    $PYTHON $INSTANCE/manage.py logparser &> /dev/null

# DNSBL checks
*/30    *       *       *       *       root    $PYTHON $INSTANCE/manage.py modo check_mx

# Public API communication
0       *       *       *       *       root    $PYTHON $INSTANCE/manage.py communicate_with_public_a
```

Now you can continue to the *Web server* section.

Web server

Note: The following instructions are meant to help you get your site up and running quickly. However it is not possible for the people contributing documentation to Modoboa to test every single combination of web server, wsgi server, distribution, etc. So it is possible that **your** installation of uwsgi or nginx or Apache or what-have-you works differently. Keep this in mind.

Apache2

First, make sure that `mod_wsgi` is installed on your server.

Create a new virtualhost in your Apache configuration and put the following content inside:

```
<VirtualHost *:80>
  ServerName <your value>
  DocumentRoot <modoboa_instance_path>

  Alias /media/ <modoboa_instance_path>/media/
  <Directory <modoboa_instance_path>/media>
    Order deny,allow
    Allow from all
  </Directory>

  Alias /sitestatic/ <modoboa_instance_path>/sitestatic/
  <Directory <modoboa_instance_path>/sitestatic>
    Order deny,allow
    Allow from all
  </Directory>

  WSGIScriptAlias / <modoboa_instance_path>/<instance_name>/wsgi.py

</VirtualHost>
```

This is just one possible configuration.

To use mod_wsgi daemon mode, add the two following directives just under `WSGIScriptAlias`:

```
WSGIDaemonProcess example.com python-path=<modoboa_instance>:<virtualenv path>/lib/python2.7/site-pac
WSGIProcessGroup example.com
```

Replace values between <> with yours. If you don't use a virtualenv, just remove the last part of the `WSGIDaemonProcess` directive.

Note: You will certainly need more configuration in order to launch Apache.

Now, you can go the *Dovecot* section to continue the installation.

Nginx

This section covers two different ways of running Modoboa behind Nginx using a WSGI application server. Choose the one you prefer between Green Unicorn or uWSGI.

In both cases, you'll need to download and install nginx.

Green Unicorn Firstly, Download and install gunicorn. Then, use the following sample gunicorn configuration (create a new file named `gunicorn.conf.py` inside Modoboa's root dir):

```
backlog = 2048
bind = "unix:/var/run/gunicorn/modoboa.sock"
pidfile = "/var/run/gunicorn/modoboa.pid"
daemon = True
debug = False
workers = 2
logfile = "/var/log/gunicorn/modoboa.log"
loglevel = "info"
```

To start gunicorn, execute the following commands:

```
$ cd <modoboa dir>
$ gunicorn -c gunicorn.conf.py <modoboa dir>.wsgi:application
```

Now the nginx part. Just create a new virtual host and use the following configuration:

```
upstream modoboa {
      server      unix:/var/run/gunicorn/modoboa.sock fail_timeout=0;
}

server {
      listen 443 ssl;
      ssl on;
      keepalive_timeout 70;

      server_name <host fqdn>;
      root <modoboa_instance_path>;

      access_log  /var/log/nginx/<host fqdn>.access.log;
      error_log /var/log/nginx/<host fqdn>.error.log;

      ssl_certificate     <ssl certificate for your site>;
      ssl_certificate_key <ssl certificate key for your site>;

      location /sitestatic/ {
              autoindex on;
      }

      location /media/ {
              autoindex on;
      }

      location / {
              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
              proxy_set_header Host $http_host;
              proxy_redirect off;
              proxy_set_header X-Forwarded-Protocol ssl;
              proxy_pass http://modoboa;
      }
}
```

If you do not plan to use SSL then change the listen directive to `listen 80;` and delete each of the following directives:

```
ssl on;
keepalive_timeout 70;
ssl_certificate     <ssl certificate for your site>;
```

```
ssl_certificate_key <ssl certificate key for your site>;
proxy_set_header X-Forwarded-Protocol ssl;
```

If you do plan to use SSL, you'll have to generate a certificate and a key. This article contains information about how to do it.

Paste this content to your configuration (replace values between <> with yours) and restart nginx.

Now, you can go the *Dovecot* section to continue the installation.

uWSGI The following setup is meant to get you started quickly. You should read the documentation of both nginx and uwsgi to understand how to optimize their configuration for your site.

The Django documentation includes the following warning regarding uwsgi:

Warning: Some distributions, including Debian and Ubuntu, ship an outdated version of uWSGI that does not conform to the WSGI specification. Versions prior to 1.2.6 do not call close on the response object after handling a request. In those cases the request_finished signal isn't sent. This can result in idle connections to database and memcache servers.

Use uwsgi 1.2.6 or newer. If you do not, you *will* run into problems. Modoboa will fail in obscure ways.

To use this setup, first download and install uwsgi.

Here is a sample nginx configuration:

```
server {
    listen 443 ssl;
    ssl on;
    keepalive_timeout 70;

    server_name <host fqdn>;
    root <modoboa's settings dir>;

    ssl_certificate     <ssl certificate for your site>;
    ssl_certificate_key <ssl certificate key for your site>;

    access_log  /var/log/nginx/<host fqdn>.access.log;
    error_log /var/log/nginx/<host fqdn>.error.log;

    location <modoboa's root url>/sitestatic/ {
            autoindex on;
            alias <location of sitestatic on your file system>;
    }

    # Whether or not Modoboa uses a media directory depends on how
    # you configured Modoboa. It does not hurt to have this.
    location <modoboa's root url>/media/ {
            autoindex on;
            alias <location of media on your file system>;
    }

    # This denies access to any file that begins with
    # ".ht". Apache's .htaccess and .htpasswd are such files. A
    # Modoboa installed from scratch would not contain any such
    # files, but you never know what the future holds.
    location ~ /\.ht {
        deny all;
    }

    location <modoba's root url>/ {
        include uwsgi_params;
        uwsgi_pass <uwsgi port>;
        uwsgi_param UWSGI_SCRIPT <modoboa instance name>.wsgi:application;
        uwsgi_param UWSGI_SCHEME https;
    }
}
```

`<modoboa instance name>` must be replaced by the value you used when you deployed your instance.

If you do not plan to use SSL then change the listen directive to `listen 80;` and delete each of the following directives:

```
ssl on;
keepalive_timeout 70;
ssl_certificate     <ssl certificate for your site>;
ssl_certificate_key <ssl certificate key for your site>;
uwsgi_param UWSGI_SCHEME https;
```

If you do plan to use SSL, you'll have to generate a certificate and a key. This article contains information about how to do it.

Make sure to replace the `<...>` in the sample configuration with appropriate values. Here are some explanations for the cases that may not be completely self-explanatory:

`<modoboa's settings dir>` Where Modoboa's `settings.py` resides. This is also where the `sitestatic` and `media` directories reside.

`<modoboa's root url>` This is the URL which will be the root of your Modoboa site at your domain. For instance, if your Modoboa installation is reachable at at `https://foo/modoboa` then `<modoboa's root url>` is `/modoboa`. In this case you probably also have to set the `alias` directives to point to where Modoboa's sitestatic and media directories are because otherwise nginx won't be able to find them.

If Modoboa is at the root of your domain, then `<modoboa root url>` is an empty string and can be deleted from the configuration above. In this case, you probably do not need the `alias` directives.

`<uwsgi port>` The location where uwsig is listening. It could be a unix domain socket or an address:port combination. Ubuntu configures uwsgi so that the port is:

```
unix:/run/uwsgi/app/<app name>/socket
```

where `<app name>` is the name of the application.

Your uwsgi configuration should be:

```
[uwsgi]
# Not needed when using uwsgi from pip
# plugins = python
chdir = <modoboa's top dir>
module = <name>.wsgi:application
master = true
harakiri = 60
processes = 4
vhost = true
no-default-app = true
```

The plugins directive should be turned on if you use a uwsgi installation that requires it. If uwsgi was installed from pip, it does not require it. In the configuration above:

<modoboa's top dir> The directory where `manage.py` resides. This directory is the parent of `<modoboa's settings dir>`

<name> The name that you passed to `modoboa-admin.py deploy` when you created your Modoboa instance.

Now, you can go the *Dovecot* section to continue the installation.

Dovecot

Modoboa requires Dovecot 2+ to work so the following documentation is suitable for this combination.

In this section, we assume dovecot's configuration resides in `/etc/dovecot`, all pathes will be relative to this directory.

Mailboxes

First, edit the `conf.d/10-mail.conf` and set the `mail_location` variable:

```
# maildir
mail_location = maildir:~/.maildir
```

Then, edit the `inbox` namespace and add the following lines:

```
inbox = yes

mailbox Drafts {
  auto = subscribe
  special_use = \Drafts
}
mailbox Junk {
  auto = subscribe
  special_use = \Junk
}
mailbox Sent {
  auto = subscribe
  special_use = \Sent
}
mailbox Trash {
  auto = subscribe
  special_use = \Trash
}
```

With dovecot 2.1+, it ensures all the special mailboxes will be automaticaly created for new accounts.

For dovecot 2.0 and older, use the autocreate plugin.

Operations on the file system

Warning: Modoboa needs to access the `dovecot` binary to check its version. To nary path, we use the `which` command first and then try known locations (`/usr/sbin/dc` `/usr/local/sbin/dovecot`). If you installed dovecot in a custom location, please tell us whe is by using the `DOVECOT_LOOKUP_PATH` setting (see `settings.py`).

Three operation types are considered:

1. Mailbox creation
2. Mailbox renaming

3. Mailbox deletion

The first one is managed by Dovecot. The last two ones may be managed by Modoboa if it can access the file system where the mailboxes are stored (see *General parameters* to activate this feature).

Those operations are treated asynchronously by a cron script. For example, when you rename an e-mail address through the web UI, the associated mailbox on the file system is not modified directly. Instead of that, a *rename* order is created for this mailbox. The mailbox will be considered unavailable until the order is not executed (see *Postfix configuration*).

Edit the crontab of the user who owns the mailboxes on the file system:

```
$ crontab -u <user> -e
```

And add the following line inside:

```
* * * * * python <modoboa_site>/manage.py handle_mailbox_operations
```

Warning: The cron script must be executed by the system user owning the mailboxes.

Warning: The user running the cron script must have access to the `settings.py` file of the modoboa instance.

The result of each order is recorded into Modoboa's log. Go to *Modoboa* > *Logs* to consult them.

Authentication

To make the authentication work, edit the `conf.d/10-auth.conf` and uncomment the following line at the end:

```
#!include auth-system.conf.ext
!include auth-sql.conf.ext
#!include auth-ldap.conf.ext
#!include auth-passwdfile.conf.ext
#!include auth-checkpassword.conf.ext
#!include auth-vpopmail.conf.ext
#!include auth-static.conf.ext
```

Then, edit the `conf.d/auth-sql.conf.ext` file and add the following content inside:

```
passdb sql {
  driver = sql
  # Path for SQL configuration file, see example-config/dovecot-sql.conf.ext
  args = /etc/dovecot/dovecot-sql.conf.ext
}

userdb sql {
  driver = sql
  args = /etc/dovecot/dovecot-sql.conf.ext
}
```

Make sure to activate only one backend (per type) inside your configuration (just comment the other ones).

Edit the `dovecot-sql.conf.ext` and modify the configuration according to your database engine.

MySQL users

```
driver = mysql

connect = host=<mysqld socket> dbname=<database> user=<user> password=<password>
```

```
default_pass_scheme = CRYPT

password_query = SELECT email AS user, password FROM core_user WHERE email='%u' and is_active=1

user_query = SELECT '<mailboxes storage directory>/%Ld/%Ln' AS home, <uid> as uid, <gid> as gid, conc

iterate_query = SELECT email AS username FROM core_user WHERE email<>''
```

PostgreSQL users

```
driver = pgsql

connect = host=<postgres socket> dbname=<database> user=<user> password=<password>

default_pass_scheme = CRYPT

password_query = SELECT email AS user, password FROM core_user WHERE email='%u' and is_active

user_query = SELECT '<mailboxes storage directory>/%Ld/%Ln' AS home, <uid> as uid, <gid> as gid, '*:b

iterate_query = SELECT email AS username FROM core_user WHERE email<>''
```

SQLite users

```
driver = sqlite

connect = <path to the sqlite db file>

default_pass_scheme = CRYPT

password_query = SELECT email AS user, password FROM core_user WHERE email='%u' and is_active=1

user_query = SELECT '<mailboxes storage directory>/%Ld/%Ln' AS home, <uid> as uid, <gid> as gid, ('*

iterate_query = SELECT email AS username FROM core_user WHERE email<>''
```

Note: Replace values between <> with yours.

LMTP

Local Mail Transport Protocol is used to let Postfix deliver messages to Dovecot.

First, make sure the protocol is activated by looking at the `protocols` setting (generally inside `dovecot.conf`). It should be similar to the following example:

```
protocols = imap pop3 lmtp
```

Then, open the `conf.d/10-master.conf`, look for `lmtp` service definition and add the following content inside:

```
service lmtp {
  # stuff before
  unix_listener /var/spool/postfix/private/dovecot-lmtp {
    mode = 0600
    user = postfix
    group = postfix
  }
  # stuff after
}
```

We assume here that Postfix is *chrooted* within `/var/spool/postfix`.

Finally, open the `conf.d/20-lmtp.conf` and modify it as follows:

```
protocol lmtp {
  postmaster_address = postmaster@<domain>
  mail_plugins = $mail_plugins quota sieve
}
```

Replace `<domain>` by the appropriate value.

Note: If you don't plan to apply quota or to use filters, just adapt the content of the `mail_plugins` setting.

Quota

Modoboa lets adminstrators define per-domain and/or per-account limits (quota). It also lists the current quota usage of each account. Those features require Dovecot to be configured in a specific way.

Inside `conf.d/10-mail.conf`, add the `quota` plugin to the default activated ones:

```
mail_plugins = quota
```

Inside `conf.d/10-master.conf`, update the `dict` service to set proper permissions:

```
service dict {
  # If dict proxy is used, mail processes should have access to its socket.
  # For example: mode=0660, group=vmail and global mail_access_groups=vmail
  unix_listener dict {
    mode = 0600
    user = <user owning mailboxes>
    #group =
  }
}
```

Inside `conf.d/20-imap.conf`, activate the `imap_quota` plugin:

```
protocol imap {
  # ...

  mail_plugins = $mail_plugins imap_quota

  # ...
}
```

Inside `dovecot.conf`, activate the quota SQL dictionary backend:

```
dict {
  quota = <driver>:/etc/dovecot/dovecot-dict-sql.conf.ext
}
```

Inside `conf.d/90-quota.conf`, activate the *quota dictionary* backend:

```
plugin {
  quota = dict:User quota::proxy::quota
}
```

It will tell Dovecot to keep quota usage in the SQL dictionary.

Finally, edit the `dovecot-dict-sql.conf.ext` file and put the following content inside:

```
connect = host=<db host> dbname=<db name> user=<db user> password=<password>
# SQLite users
# connect = /path/to/the/database.db

map {
  pattern = priv/quota/storage
  table = admin_quota
  username_field = username
  value_field = bytes
}
map {
  pattern = priv/quota/messages
  table = admin_quota
  username_field = username
  value_field = messages
}
```

PostgreSQL users

Database schema update The `admin_quota` table is created by Django but unfortunately it doesn't support `DEFAULT` constraints (it only simulates them when the ORM is used). As PostgreSQL is a bit strict about constraint violations, you must execute the following query manually:

```
db=> ALTER TABLE admin_quota ALTER COLUMN bytes SET DEFAULT 0;
db=> ALTER TABLE admin_quota ALTER COLUMN messages SET DEFAULT 0;
```

Trigger As indicated on Dovecot's wiki, you need a trigger to properly update the quota.

A working copy of this trigger is available on Modoboa's website.

Download this file and copy it on the server running postgres. Then, execute the following commands:

```
$ su - postgres
$ psql [modoboa database] < /path/to/modoboa_postgres_trigger.sql
$ exit
```

Replace `[modoboa database]` by the appropriate value.

Forcing recalculation For existing installations, *Dovecot* (> 2) offers a command to recalculate the current quota usages. For example, if you want to update all usages, run the following command:

```
$ doveadm quota recalc -A
```

Be carefull, it can take a while to execute.

ManageSieve/Sieve

Modoboa lets users define filtering rules from the web interface. To do so, it requires *ManageSieve* to be activated on your server.

Inside `conf.d/20-managesieve.conf`, make sure the following lines are uncommented:

```
protocols = $protocols sieve

service managesieve-login {
  # ...
}

service managesieve {
  # ...
}

protocol sieve {
  # ...
}
```

Messages filtering using Sieve is done by the LDA.

Inside `conf.d/15-lda.conf`, activate the `sieve` plugin like this:

```
protocol lda {
  # Space separated list of plugins to load (default is global mail_plugins).
  mail_plugins = $mail_plugins sieve
}
```

Finally, configure the `sieve` plugin by editing the `conf.d/90-sieve.conf` file. Put the follwing caontent inside:

```
plugin {
  # Location of the active script. When ManageSieve is used this is actually
  # a symlink pointing to the active script in the sieve storage directory.
  sieve = ~/.dovecot.sieve

  #
  # The path to the directory where the personal Sieve scripts are stored. For
  # ManageSieve this is where the uploaded scripts are stored.
  sieve_dir = ~/sieve
}
```

Restart Dovecot.

Now, you can go to the *Postfix* section to finish the installation.

Postfix

This section gives an example about building a simple virtual hosting configuration with *Postfix*. Refer to the official documentation for more explanation.

Map files

You first need to create configuration files (or map files) that will be used by Postfix to lookup into Modoboa tables.

To automaticaly generate the requested map files and store them in a directory, run the following command:

```
> cd <modoboa_instance_path>
> python manage.py generate_postfix_maps --destdir <directory>
```

`<directory>` is the directory where the files will be stored. Answer the few questions and you're done.

Configuration

Use the following configuration in the `/etc/postfix/main.cf` file (this is just one possible configuration):

```
# Stuff before
virtual_transport = lmtp:unix:private/dovecot-lmtp

relay_domains =
virtual_mailbox_domains = <driver>:/etc/postfix/sql-domains.cf
virtual_alias_domains = <driver>:/etc/postfix/sql-domain-aliases.cf
virtual_alias_maps = <driver>:/etc/postfix/sql-aliases.cf

relay_domains = <driver>:/etc/postfix/sql-relaydomains.cf
transport_maps =
    <driver>:/etc/postfix/sql-spliteddomains-transport.cf
    <driver>:/etc/postfix/sql-relaydomains-transport.cf

smtpd_recipient_restrictions =
      # ...
      check_recipient_access
          <driver>:/etc/postfix/sql-maintain.cf
          <driver>:/etc/postfix/sql-relay-recipient-verification.cf
      permit_mynetworks
      reject_unauth_destination
      reject_unverified_recipient
      # ...

smtpd_sender_login_maps =
      <driver>:/etc/postfix/sql-sender-login-mailboxes.cf
      <driver>:/etc/postfix/sql-sender-login-aliases.cf
      <driver>:/etc/postfix/sql-sender-login-mailboxes-extra.cf

smtpd_sender_restrictions =
      reject_sender_login_mismatch

# Stuff after
```

Replace `<driver>` by the name of the database you use.

Restart Postfix.

Extensions

Only few commands are needed to add a new extension to your setup.

In case you use a dedicated user and/or a virtualenv, do not forget to use them:

```
> sudo -i <modoboa_user>
> source <virtuenv_path>/bin/activate
```

Then, run the following commands:

```
> pip install <EXTENSION>==<VERSION>
> cd <modoboa_instance_dir>
> python manage.py migrate
> python manage.py collectstatic
```

Then, restart your web sever.

2.2 Upgrade

2.2.1 Modoboa

Warning: The new version you are going to install may need to modify your database. Before you start, make sure to backup everything!

Most of the time, upgrading your installation to a newer Modoboa version only requires a few actions. In every case, you will need to apply the general procedure first and then check if the version you are installing requires specific actions.

In case you use a dedicated user and/or a virtualenv, do not forget to use them:

```
> sudo -i <modoboa_user>
> source <virtuenv_path>/bin/activate
```

Then, run the following commands:

```
> pip install modoboa==<VERSION>
> cd <modoboa_instance_dir>
> python manage.py migrate
> python manage.py collectstatic
```

Once done, check if the version you are installing requires *Specific instructions*.

Finally, restart your web server.

Sometimes, you might need to upgrade postfix map files too. To do so, just run the `generate_postfix_maps` command on the same directory than the one used for installation (`/etc/postfix` by default).

Make sure to use root privileges and run the following command:

```
> python manage.py generate_postfix_maps --destdir <directory>
```

Then, reload postfix.

2.2.2 Extensions

If a new version is available for an extension you're using, it is recommanded to install it. Upgrading an extensions is pretty and the procedure is almost the same than the one used for Modoboa.

In case you use a dedicated user and/or a virtualenv, do not forget to use them:

```
> sudo -i <modoboa_user>
> source <virtuenv_path>/bin/activate
```

Then, run the following commands:

```
> pip install <EXTENSION>==<VERSION>
> cd <modoboa_instance_dir>
> python manage.py migrate
> python manage.py collectstatic
```

Finally, restart your web server.

It is a generic upgrade procedure which will be enough most of the time but it is generally a good idea to check the associated documentation.

2.2.3 Specific instructions

1.6.0

An interesting feature brougth by this version is the capability to make different checks about MX records. For example, Modoboa can query main DNSBL providers for every defined domain. With this, you will quickly know if one the domains you manage is listed or not. To activate it, add the following line to your crontab:

```
*/30 * * * * <optional_virtualenv_path/>python <modoboa_instance_dir>/manage.py modo check_mx
```

The communication with Modoboa public API has been reworked. Instead of sending direct synchronous queries (for example to check new versions), a cron job has been added. To activate it, add the following line to your crontab:

```
0 * * * * <optional_virtualenv_path/>python <modoboa_instance_dir>/manage.py communicate_with_public_
```

Please also note that public API now uses TLS so you must update your configuration as follows:

```
MODOBOA_API_URL = 'https://api.modoboa.org/1/'
```

Finally, it is now possible to declare additional sender addresses on a per-account basis. You need to update your postfix configuration in order to use this functionality. Just edit the `main.cf` file and change the following parameter:

```
smtpd_sender_login_maps =
    <driver>:/etc/postfix/sql-sender-login-mailboxes.cf
    <driver>:/etc/postfix/sql-sender-login-aliases.cf
    <driver>:/etc/postfix/sql-sender-login-mailboxes-extra.cf
```

1.5.0

The API has been greatly improved and a documentation is now available. To enable it, add `'rest_framework_swagger'` to the `INSTALLED_APPS` variable in `settings.py` as follows:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.sites',
    'django.contrib.staticfiles',
    'reversion',
    'rest_framework.authtoken',
    'rest_framework_swagger',
)
```

Then, add the following content into `settings.py`, just after the `REST_FRAMEWORK` variable:

```
SWAGGER_SETTINGS = {
    "is_authenticated": False,
    "api_version": "1.0",
    "exclude_namespaces": [],
    "info": {
        "contact": "contact@modoboa.com",
        "description": ("Modoboa API, requires a valid token."),
        "title": "Modoboa API",
    }
}
```

You're done. The documentation is now available at the following address:

> http://<your instance address>/docs/api/

Finally, if you find a `TEMPLATE_CONTEXT_PROCESSORS` variable in your `settings.py` file, make sure it looks like this:

```
TEMPLATE_CONTEXT_PROCESSORS = global_settings.TEMPLATE_CONTEXT_PROCESSORS + [
    'modoboa.core.context_processors.top_notifications',
]
```

1.4.0

Warning: Please make sure to use Modoboa 1.3.5 with an up-to-date database before an upgrade to 1.4.0.

Warning: Do not follow the regular upgrade procedure for this version.

Some extension have been moved back into the main repository. The main reason for that is that using Modoboa without them doesn't make sense.

First of all, you must rename the following applications listed inside the `MODOBOA_APPS` variable:

Old name	New name
modoboa_admin	modoboa.admin
modoboa_admin_limits	modoboa.limits
modoboa_admin_relaydomains	modoboa.relaydomains

Then, apply the following steps:

1. Uninstall old extensions:

```
   $ pip uninstall modoboa-admin modoboa-admin-limits modoboa-admin-relaydomains
```

2. Install all extension updates using pip (check the *Modoboa* > *Information* page)
3. Manually migrate database:

```
   $ cd <instance_dir>
   $ python manage.py migrate auth
   $ python manage.py migrate admin 0001 --fake
   $ python manage.py migrate admin
   $ python manage.py migrate limits 0001 --fake
   $ python manage.py migrate relaydomains 0001 --fake
   $ python manage.py migrate
```

4. Finally, update static files:

```
$ python manage.py collectstatic
```

This version also introduces a REST API. To enable it:

1. Add `'rest_framework.authtoken'` to the `INSTALLED_APPS` variable
2. Add the following configuration inside `settings.py`:

```
# Rest framework settings

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.TokenAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    )
}
```

3. Run the following command:

```
$ python manage.py migrate
```

1.3.5

To enhance security, Modoboa now checks the *strength of user passwords <https://github.com/dstufft/django-passwords>*_.

To use this feature, add the following configuration into the `settings.py` file:

```
# django-passwords

PASSWORD_MIN_LENGTH = 8

PASSWORD_COMPLEXITY = {
    "UPPER": 1,
    "LOWER": 1,
    "DIGITS": 1
}
```

1.3.2

Modoboa now uses the *atomic requests* mode to preserve database consistency (reference).

To enable it, update the `DATABASES` variable in `settings.py` as follows:

```
DATABASES = {
    "default": {
        # stuff before...
        "ATOMIC_REQUESTS": True
    },
    "amavis": {
        # stuff before...
        "ATOMIC_REQUESTS": True
    }
}
```

1.3.0

This release does not bring awesome new features but it is a necessary bridge to the future of Modoboa. All extensions now have their own git repository and the deploy process has been updated to reflect this change.

Another important update is the use of Django 1.7. Besides its new features, the migration system has been reworked and is now more robust than before.

Before we begin with the procedure, here is a table showing old extension names and their new name:

Old name	New package name	New module name
modoboa.extensions.admin	modoboa-admin	modoboa_admin
modoboa.extensions.limits	modoboa-admin-limits	modoboa_admin_limits
modoboa.extensions.postfix_autoreply	modoboa-postfix-autoreply	modoboa_postfix_autoreply
modoboa.extensions.postfix_relay_domains	modoboa-admin-relaydomains	modoboa_admin_relaydomains
modoboa.extensions.radicale	modoboa-radicale	modoboa_radicale
modoboa.extensions.sievefilters	modoboa-sievefilters	modoboa_sievefilters
modoboa.extensions.stats	modoboa-stats	modoboa_stats
modoboa.extensions.webmail	modoboa-webmail	modoboa_webmail

Here are the required steps:

1. Install the extensions using pip (look at the second column in the table above):

```
$ pip install <the extensions you want>
```

2. Remove `south` from `INSTALLED_APPS`
3. Rename old extension names inside `MODOBOA_APPS` (look at the third column in the table above)
4. Remove `modoboa.lib.middleware.ExtControlMiddleware` from `MIDDLEWARE_CLASSES`
5. Change `DATABASE_ROUTERS` to:

```
DATABASE_ROUTERS = ["modoboa_amavis.dbrouter.AmavisRouter"]
```

6. Run the following commands:

```
$ cd <modoboa_instance_dir>
$ python manage.py migrate
```

7. Reply `yes` to the question
8. Run the following commands:

```
$ python manage.py load_initial_data
$ python manage.py collectstatic
```

9. The cleanup job has been renamed in Django, so you have to modify your crontab entry:

```
- 0 0 * * * <modoboa_site>/manage.py cleanup
```

   - 0 0 * * * <modoboa_site>/manage.py clearsessions

1.2.0

A new notification service let administrators know about new Modoboa versions. To activate it, you need to update the `TEMPLATE_CONTEXT_PROCESSORS` variable like this:

```
from django.conf import global_settings

TEMPLATE_CONTEXT_PROCESSORS = global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
  'modoboa.core.context_processors.top_notifications',
)
```

and to define the new `MODOBOA_API_URL` variable:

```
MODOBOA_API_URL = 'http://api.modoboa.org/1/'
```

The location of external static files has changed. To use them, add a new path to the `STATICFILES_DIRS`:

```
# Additional locations of static files
STATICFILES_DIRS = (
  # Put strings here, like "/home/html/static" or "C:/www/django/static".
  # Always use forward slashes, even on Windows.
  # Don't forget to use absolute paths, not relative paths.
  "<path/to/modoboa/install/dir>/bower_components",
)
```

Run the following commands to define the hostname of your instance:

```
$ cd <modoboa_instance_dir>
$ python manage.py set_default_site <hostname>
```

If you plan to use the Radicale extension:

1. Add `'modoboa.extensions.radicale'` to the `MODOBOA_APPS` variable
2. Run the following commands:

```
    $ cd <modoboa_instance_dir>
    $ python manage.py syncdb
```

Warning: You also have to note that the `sitestatic` directory has moved from `<path to your site's dir>` to `<modoboa's root url>` (it's probably the parent directory). You have to adapt your web server configuration to reflect this change.

2.3 Configuration

2.3.1 Online parameters

Modoboa provides online panels to modify internal parameters. There are two available levels:

- Application level: global parameters, define how the application behaves. Available at *Modoboa* > *Parameters*
- User level: per user customization. Available at *User* > *Settings* > *Preferences*

Regardless level, parameters are displayed using tabs, each tab corresponding to one application.

General parameters

The *admin* application exposes several parameters, they are presented below:

Name	Description	Default value
Authentication type	The backend used for authentication	Local
Default password scheme	Scheme used to crypt mailbox passwords	crypt
Secret key	A key used to encrypt users' password in sessions	random value
Handle mailboxes on filesystem	Rename or remove mailboxes on the filesystem when they get renamed or removed within Modoboa	no
Mailboxes owner	The UNIX account who owns mailboxes on the filesystem	vmail
Automatic account removal	When a mailbox is removed, also remove the associated account	no
Maximum log record age	The maximum age in days of a log record	365
Items per page	Number of displayed items per page	30
Default top redirection	The default redirection used when no application is specified	userprefs

Note: If you are not familiar with virtual domain hosting, you should take a look at postfix's documentation. This How to also contains useful information.

Note: A random secret key will be generated each time the *Parameters* page is refreshed and until you save parameters at least once.

Note: Specific LDAP parameters are also available, see *LDAP authentication*.

2.3.2 Media files

Modoboa uses a specific directory to upload files (ie. when the webmail is in use) or to create ones (ex: graphical statistics). This directory is named `media` and is located inside modoboa's installation directory (called `modoboa_site` in this documentation).

To work properly, the system user which runs modoboa (`www-data`, `apache`, whatever) must have write access to this directory.

2.3.3 Customization

Custom logo

You have the possibility to use a custom logo instead of the default one on the login page.

To do so, open the `settings.py` file and add a `MODOBOA_CUSTOM_LOGO` variable. This variable must contain the relative URL of your logo under `MEDIA_URL`. For example:

```
MODOBOA_CUSTOM_LOGO = os.path.join(MEDIA_URL, "custom_logo.png")
```

Then copy your logo file into the directory indicated by `MEDIA_ROOT`.

2.3.4 Host configuration

Note: This section is only relevant when Modoboa handles mailboxes renaming and removal from the filesystem.

To manipulate mailboxes on the filesystem, you must allow the user who runs Modoboa to execute commands as the user who owns mailboxes.

To do so, edit the /etc/sudoers file and add the following inside:

```
<user_that_runs_modoboa> ALL=(<mailboxes owner>) NOPASSWD: ALL
```

Replace values between <> by the ones you use.

2.3.5 Time zone and language

Modoboa is available in many languages.

To specify the default language to use, edit the settings.py file and modify the LANGUAGE_CODE variable:

```
LANGUAGE_CODE = 'fr' # or 'en' for english, etc.
```

Note: Each user has the possibility to define the language he prefers.

In the same configuration file, specify the timezone to use by modifying the TIME_ZONE variable. For example:

```
TIME_ZONE = 'Europe/Paris'
```

2.3.6 Sessions management

Modoboa uses Django's session framework to store per-user information.

Few parameters need to be set in the settings.py configuration file to make Modoboa behave as expected:

```
SESSION_EXPIRE_AT_BROWSER_CLOSE = False # Default value
```

This parameter is optional but you must ensure it is set to False (the default value).

The default configuration file provided by the modoboa-admin.py command is properly configured.

2.3.7 LDAP

Authentication

Modoboa supports external LDAP authentication using the following extra components:

- Python LDAP client
- Django LDAP authentication backend

If you want to use this feature, you must first install those components:

```
$ pip install python-ldap django-auth-ldap
```

Then, all you have to do is to modify the `settings.py` file. Add a new authentication backend to the *AUTHENTICATION_BACKENDS* variable, like this:

```
AUTHENTICATION_BACKENDS = (
  'modoboa.lib.authbackends.LDAPBackend',
  'modoboa.lib.authbackends.SimpleBackend',
)
```

Finally, go to *Modoboa* > *Parameters* > *General* and set *Authentication type* to LDAP.

From there, new parameters will appear to let you configure the way Modoboa should connect to your LDAP server. They are described just below:

Name	Description	Default value
Server address	The IP address of the DNS name of the LDAP server	local-host
Server port	The TCP port number used by the LDAP server	389
Use a secure connection	Use an SSL/TLS connection to access the LDAP server	no
Authentication method	Choose the authentication method to use	Direct bind
User DN template (direct bind mode)	The template used to construct a user's DN. It should contain one placeholder (ie. `%(user)s`)	
Bind BN	The distinguished name to use when binding to the LDAP server. Leave empty for an anonymous bind	
Bind password	The password to use when binding to the LDAP server (with 'Bind DN')	
Search base	The distinguished name of the search base	
Search filter	An optional filter string (e.g. '(objectClass=person)'). In order to be valid, it must be enclosed in parentheses.	(mail=%(user)s)
Password attribute	The attribute used to store user passwords	user-Pass-word
Active Directory	Tell if the LDAP server is an Active Directory one	no
Administrator groups	Members of those LDAP Posix groups will be created ad domain administrators. Use ';' characters to separate groups.	
Group type	The type of group used by your LDAP directory.	Posix-Group
Groups search base	The distinguished name of the search base used to find groups	
Domain/mailbox creation	Automatically create a domain and a mailbox when a new user is created just after the first successful authentication. You will generally want to disable this feature when the relay domains extension is in use	yes

If you need additional parameters, you will find a detailled documentation here.

Once the authentication is properly configured, the users defined in your LDAP directory will be able to connect to Modoboa, the associated domain and mailboxes will be automatically created if needed.

The first time a user connects to Modoboa, a local account is created if the LDAP username is a valid email address. By default, this account belongs to the *SimpleUsers* group and it has a mailbox.

To automatically create domain administrators, you can use the **Administrator groups** setting. If a LDAP user belongs to one the listed groups, its local account will belong to the *DomainAdmins* group. In this case, the username is not necessarily an email address.

Users will also be able to update their LDAP password directly from Modoboa.

Note: Modoboa doesn't provide any synchronization mechanism once a user is registered into the database. Any modification done from the directory to a user account will not be reported to Modoboa (an email address change for example). Currently, the only solution is to manually delete the Modoboa record, it will be recreated on the next user login.

2.3.8 Database maintenance

Cleaning the logs table

Modoboa logs administrator specific actions into the database. A clean-up script is provided to automatically remove oldest records. The maximum log record age can be configured through the online panel.

To use it, you can setup a cron job to run every night:

```
0 0 * * * <modoboa_site>/manage.py cleanlogs
#
# Or like this if you use a virtual environment:
# 0 0 * * * <virtualenv path/bin/python> <modoboa_site>/manage.py cleanlogs
```

Cleaning the session table

Django does not provide automatic purging. Therefore, it's your job to purge expired sessions on a regular basis.

Django provides a sample clean-up script: `django-admin.py clearsessions`. That script deletes any session in the session table whose `expire_date` is in the past.

For example, you could setup a cron job to run this script every night:

```
0 0 * * * <modoboa_site>/manage.py clearsessions
#
# Or like this if you use a virtual environment:
# 0 0 * * * <virtualenv path/bin/python> <modoboa_site>/manage.py clearsessions
```

2.4 REST API

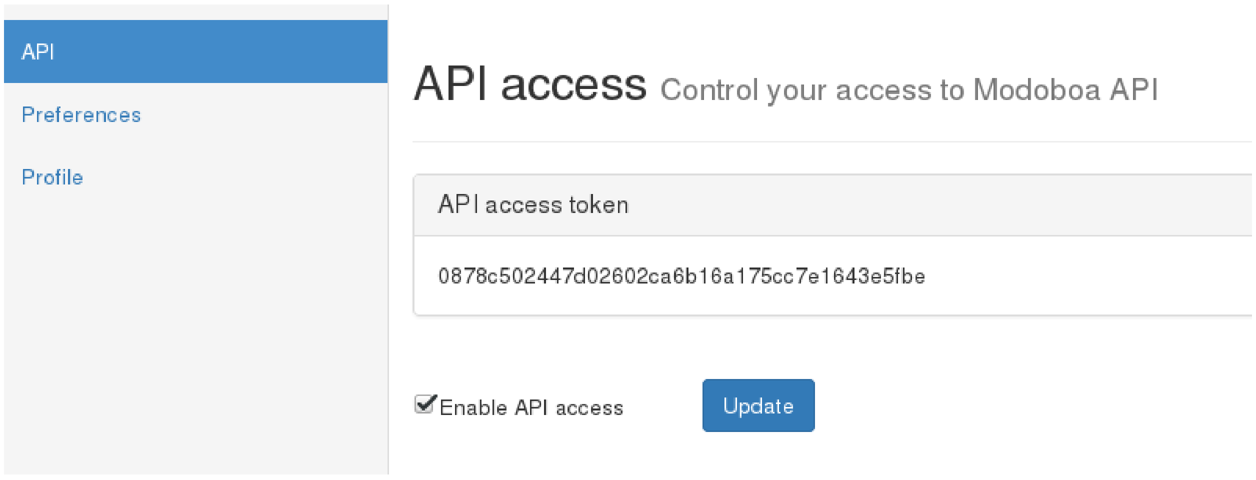
To ease the integration with external sources (software or other), Modoboa provides a REST API.

Every installed instance comes with a ready-to-use API and a documentation. You will find them using the following url patterns:

- API: *http://<hostname>/api/v1/*
- Documentation: *http://<hostname>/docs/api/*

An example of this documentation is available on the official demo.

Using this API requires an authentication and for now, only a token based authentication is supported. To get a valid token, log-in to your instance with a super administrator, go to *Settings > API* and activate the API access. Press the Update button and wait until the page is reloaded, the token will be displayed.


API
Preferences
Profile
API access Control your access to Modoboa API
API access token
0878c502447d02602ca6b16a175cc7e1643e5fbe
Enable API access
Update

To make valid API calls, every requests you send must embed this token within an Authorization HTTP header like this:

```
Authorization: Token <YOUR_TOKEN>
```

and the content type of those requests must be `application/json`.

2.5 How to contribute

Contributions are always welcome. If you want to submit a patch, please respect the following rules:

- Open a pull request on the appropriate repository
- Respect PEP8
- Document your patch and respect PEP 257
- Add unit tests and make sure the global coverage does not decrease

If all those steps are validated, your contribution will generally be integrated.

2.5.1 Table of contents

Useful tips

You would like to work on Modoboa but you don't know where to start? You're at the right place! Browse this page to learn useful tips.

Prepare a virtual environment

A virtual environment is a good way to setup a development environment on your machine.

Note: `virtualenv` is available on all major distributions, just install it using your favorite packages manager.

To do so, run the following commands:

```
$ virtualenv <path>
$ source <path>/bin/activate
$ git clone https://github.com/tonioo/modoboa.git
$ cd modoboa
$ python setup.py develop
```

The `develop` command creates a symbolic link to your local copy so any modification you make will be automatically available in your environment, no need to copy them.

Deploy an instance for development

Warning: Make sure to *create a database* before running this step.

Now that you have a *running environment*, you're ready to deploy a test instance:

```
$ modoboa-admin.py deploy --devel --dburl <database url> <path>
$ cd <path>
$ python manage.py runserver
```

You're ready to go!

Manage static files

Modoboa uses bower (thanks to django-bower) to manage its CSS and javascript dependencies.

Those dependencies are listed in a file called `dev_settings.py` located inside the `<path_to_local_copy>/modoboa/core` directory.

If you want to add a new dependency, just complete the `BOWER_INSTALLED_APPS` parameter and run the following command:

```
$ python manage.py bower install
```

It will download and store the required files into the `<path_to_local_copy>/modoboa/bower_components` directory.

Test your modifications

If you deployed a specific instance for your development needs, you can run the tests suite as follows:

```
> python manage.py test modoboa.core modoboa.lib modoboa.admin modoboa.limits modoboa.relaydomains
```

Otherwise, you can run the tests suite from the repository using tox.

Start a basic Modoboa instance

From the repository, run the following command to launch a simple instance with a few fixtures:

```
> tox -e serve
```

You can use admin/password to log in.

Build the documentation

If you need to modify the documenation and want to see the result, you can build it as follows:

```
> tox -e doc
> firefox .tox/doc/tmp/html/index.html
```

FAQ

bower command is missing in manage.py *bower* command is missing in *manage.py* if you don't use the `--devel` option of the `modoboa-admin.py deploy` command.

To fix it, regenerate your instance or update your `settings.py` file manually. Look at `devmode` in https://github.com/tonioo/modoboa/blob/master/modoboa/core/commands/templates/settings.py.tpl

Create a new plugin

Introduction

Modoboa offers a plugin API to expand its capabilities. The current implementation provides the following possibilities:

- Expand navigation by adding entry points to your plugin inside the GUI
- Access and modify administrative objects (domains, mailboxes, etc.)
- Register callback actions for specific events

Plugins are nothing more than Django applications with an extra piece of code that integrates them into Modoboa. The `modo_extension.py` file will contain a complete description of the plugin:

- Admin and user parameters
- Observed events
- Custom menu entries

The communication between both applications is provided by *Available events*. Modoboa offers some kind of hooks to let plugins add custom actions.

The following subsections describe the plugin architecture and explain how you can create your own.

The required glue

To create a new plugin, just start a new django application like this (into Modoboa's directory):

```
$ python manage.py startapp
```

Then, you need to register this application using the provided API. Just copy/paste the following example into the `modo_extension.py` file of the future extension:

```
from modoboa.core.extensions import ModoExtension, exts_pool


class MyExtension(ModoExtension):
    """My custom Modoboa extension."""

    name = "myext"
    label = "My Extension"
    version = "0.1"
    description = "A description"
    url = "myext_root_location" # optional, name is used if not defined

    def load(self):
        """This method is called when Modoboa loads available and activated plugins.

        Declare parameters and register events here.
        """
        pass

    def load_initial_data(self):
        """Optional: provide initial data for your extension here."""
        pass

exts_pool.register_extension(MyExtension)
```

Once done, simply add your extension's module name to the `MODOBOA_APPS` variable located inside `settings.py`. Finally, run the following commands:

```
$ python manage.py migrate
$ python manage.py load_initial_data
$ python manage.py collectstatic
```

Parameters

A plugin can declare its own parameters. There are two levels available:

- 'Administration' parameters : used to configure the plugin, editable inside the *Admin* > *Settings* > *Parameters* page
- 'User' parameters : per-user parameters (or preferences), editable inside the *Options* > *Preferences* page

Playing with parameters To declare a new administration parameter, use the following function:

```
from modoboa.lib import parameters

parameters.register_admin(name, **kwargs)
```

To declare a new user parameter, use the following function:

```
parameter.register_user(name, **kwargs)
```

Both functions accept extra arguments listed here:

- `type` : parameter's type, possible values are : `int`, `string`, `list`, `list_yesno`,
- `deflt` : default value,
- `help` : help text,
- `values` : list of possible values if `type` is `list`.

Custom administrative roles

Modoboa uses Django's internal permission system. Administrative roles are nothing more than groups (`Group` instances).

If an extension needs to add new roles, the following steps are required:

1. Listen to the *GetExtraRoles* event that will return the group's name
2. Listen to the *GetExtraRolePermissions* event that will return the new group's permissions

The group will automatically be created the next time you run the `load_initial_data` command.

Extending admin forms

the forms used to edit objects (account, domain, etc.) through the admin panel are composed of tabs. You can extend those forms (ie. add new tabs) in a pretty easy way by defining events.

Account To add a new tab to the account edition form, define new listeners (handlers) for the following events:

- *ExtraAccountForm*
- *FillAccountInstances*
- *CheckExtraAccountForm* (optional)

Example:

```
from modoboa.lib import events

@events.observe("ExtraAccountForm")
def extra_account_form(user, account=None):
    return [
        {"id": "tabid", "title": "Title", "cls": MyFormClass}
    ]

@events.observe("FillAccountInstances")
def fill_my_tab(user, account, instances):
    instances["id"] = my_instance
```

Domain To add a new tab to the domain edition form, define new listeners (handlers) for the following events:

- *ExtraDomainForm*
- *FillDomainInstances*

Example:

```
from modoboa.lib import events

@events.observe("ExtraDomainForm")
def extra_domain_form(user, domain):
    return [
        {"id": "tabid", "title": "Title", "cls": MyFormClass}
    ]

@events.observe("FillDomainInstances")
def fill_my_tab(user, domain, instances):
    instances["id"] = my_instance
```

Available events

Introduction

Modoboa provides a simple API to interact with events. It understands two kinds of events:

- Those returning a value
- Those returning nothing

Listening to a specific event is achieved as follows:

```
from modoboa.lib import events

def callback(*args):
  pass

events.register('event', callback)
```

You can also use the custom decorator `events.observe`:

```
@events.observe('event')
def callback(*args):
  pass
```

`event` is the event's name you want to listen to, `callback` is the function that will be called each time this event is raised. Each event impose to callbacks a specific prototype to respect. See below for a detailled list.

To stop listening to as specific event, you must use the `unregister` function:

```
events.unregister('event', callback)
```

The parameters are the same than those used with `register`.

To unregister all events declared by a specific extension, use the `unregister_extension` function:

```
events.unregister_extension([name])
```

`name` is the extension's name but it is optional. Leave it empty to let the function guess the name.

Read further to get a complete list and description of all available events.

Supported events

AccountAutoCreated Raised when a new account is automatically created (example: LDAP synchronization).

Callback prototype:

```
def callback(account): pass
```

- `account` is the newly created account (`User` instance)

AccountCreated Raised when a new account is created.

Callback prototype:

```
def callback(account): pass
```

- `account` is the newly created account (`User` instance)

AccountDeleted Raised when an existing account is deleted.

Callback prototype:

```
def callback(account, byuser, **options): pass
```

- `account` is the account that is going to be deleted
- `byuser` is the adminstrator deleting `account`

AccountExported Raised when an account is exported to CSV.

Callback prototype:

```
def callback(account): pass
```

- `account` is the account being exported

Must return a list of values to include in the export.

AccountImported Raised when an account is imported from CSV.

Callback prototype:

```
def callback(user, account, row): pass
```

- `user` is the user importing the account
- `account` is the account being imported
- `row` is a list containing what remains from the CSV definition

AccountModified Raised when an existing account is modified.

Callback prototype:

```
def callback(oldaccount, newaccount): pass
```

- `oldaccount` is the account before it is modified
- `newaccount` is the account after the modification

AdminMenuDisplay Raised when an admin menu is about to be displayed.

Callback prototype:

```
def callback(target, user): pass
```

The `target` argument indicates which menu is being displayed. Possible values are:

- `admin_menu_box` : corresponds to the menu bar available inside administration pages
- `top_menu` : corresponds to the top black bar

See *UserMenuDisplay* for a description of what callbacks that listen to this event must return.

CheckDomainName Raised before the unicity of a domain name is checked. By default, modoboa prevents duplicate names between domains and domain aliases but extensions have the possibility to extend this rule using this event.

Callback prototype:

```
def callback(): pass
```

Must return a list of 2uple, each one containing a model class and an associated label.

CheckExtraAccountForm When an account is being modified, this event lets extensions check if this account is concerned by a specific form.

Callback prototype:

```
def callback(account, form): pass
```

- `account` is the `User` instance beeing modified
- `form` is a dictionnary (same content as for `ExtraAccountForm`)

Callbacks listening to this event must return a list containing one Boolean.

DomainAliasCreated Raised when a new domain alias is created.

Callback prototype:

```
def callback(user, domain_alias): pass
```

- `user` is the new domain alias owner (`User` instance)
- `domain_alias` is the new domain alias (`DomainAlias` instance)

DomainAliasDeleted Raised when an existing domain alias is about to be deleted.

Callback prototype:

```
def callback(domain_alias): pass
```

- `domain_alias` is a `DomainAlias` instance

DomainCreated Raised when a new domain is created.

Callback prototype:

```
def callback(user, domain): pass
```

- `user` corresponds to the `User` object creating the domain (its owner)
- `domain` is a `Domain` instance

DomainDeleted Raised when an existing domain is about to be deleted.

Callback prototype:

```
def callback(domain): pass
```

- `domain` is a `Domain` instance

DomainModified Raised when a domain has been modified.

Callback prototype:

```
def callback(domain): pass
```

- `domain` is the modified `Domain` instance, it contains an extra `oldname` field which contains the old name

DomainOwnershipRemoved Raised before the ownership of a domain is removed from its original creator.

Callback prototype:

```
def callback(owner, domain): pass
```

- `owner` is the original creator
- `domain` is the `Domain` instance being modified

ExtraAccountActions Raised when the account list is displayed. Let developers define new actions to act on a specific user.

Callback prototype:

```
def callback(account): pass
```

- `account` is the account being listed

ExtraAccountForm Let extensions add new forms to the account edition form (the one with tabs).

Callback prototype:

```
def callback(user, account): pass
```

- `user` is a `User` instance corresponding to the currently logged in user
- `account` is the account beeing modified (`User` instance)

Callbacks listening to the event must return a list of dictionnaries, each one must contain at least three keys:

```
{"id" : "<the form's id>",
 "title" : "<the title used to present the form>",
 "cls" : TheFormClassName}
```

ExtraAdminContent Let extensions add extra content into the admin panel.

Callback prototype:

```
def callback(user, target, currentpage): pass
```

- `user` is a `User` instance corresponding to the currently logged in user
- `target` is a string indicating the place where the content will be displayed. Possible values are : `leftcol`
- `currentpage` is a string indicating which page (or section) is displayed. Possible values are : `domains`, `identities`

Callbacks listening to this event must return a list of string.

ExtraDomainEntries Raised to request extra entries to display inside the *domains* listing.

Callback prototype:

```
def callback(user, domfilter, searchquery, **extrafilters): pass
```

- `user` is the `User` instance corresponding to the currently logged in user
- `domfilter` is a string indicating which domain type the user needs
- `searchquery` is a string containing a search query
- `extrafilters` is a set of keyword arguments that may contain additional filters

Must return a valid `QuerySet`.

ExtraDomainFilters Raised to request extra filters for the *domains* listing page. For example, the *postfix_relay_domains* extension let users filter entries based on service types.

Callback prototype:

```
def callback(): pass
```

Must return a list of valid filter names (string).

ExtraDomainForm Let extensions add new forms to the domain edition form (the one with tabs).

Callback prototype:

```
def callback(user, domain): pass
```

- `user` is a `User` instance corresponding to the currently logged in user
- `domain` is the domain beeing modified (`Domain` instance)

Callbacks listening to the event must return a list of dictionnaries, each one must contain at least three keys:

```
{"id" : "<the form's id>",
 "title" : "<the title used to present the form>",
 "cls" : TheFormClassName}
```

ExtraDomainImportHelp Raised to request extra help text to display inside the domain import form.

Callback prototype:

```
def callback(): pass
```

Must return a list a string.

ExtraDomainMenuEntries Raised to request extra entries to include in the left menu of the *domains* listing page.

Callback prototype:

```
def callback(user): pass
```

- `user` is the `User` instance corresponding to the currently logged in user

Must return a list of dictionaries. Each dictionary must contain at least three keys:

```
{"name": "<menu name>",
 "label": "<menu label>",
 "url": "<menu url>"}
```

ExtraFormFields Raised to request extra fields to include in a django form.

Callback prototype:

```
def callback(form_name, instance=None): pass
```

- `form_name` is a string used to distinguish a specific form
- `instance` is a django model instance related to `form_name`

Must return a list of 2uple, each one containing the following information:

```
('field name', <Django form field instance>)
```

ExtraRelayDomainForm Let extensions add new forms to the relay domain edition form (the one with tabs).

Callback prototype:

```
def callback(user, rdomain): pass
```

- `user` is the `User` instance corresponding to the currently logged in user
- `rdomain` is the relay domain being modified (`RelayDomain` instance)

Callbacks listening to the event must return a list of dictionnaries, each one must contain at least three keys:

```
{"id" : "<the form's id>",
 "title" : "<the title used to present the form>",
 "cls" : TheFormClassName}
```

FillAccountInstances When an account is beeing modified, this event is raised to fill extra forms.

Callback prototype:

```
def callback(user, account, instances): pass
```

- `user` is a `User` instance corresponding to the currently logged in user
- `account` is the `User` instance beeing modified
- `instances` is a dictionnary where the callback will add information needed to fill a specific form

FillDomainInstances When a domain is beeing modified, this event is raised to fill extra forms.

Callback prototype:

```
def callback(user, domain, instances): pass
```

- `user` is a `User` instance corresponding to the currently logged in user
- `domain` is the `Domain` instance beeing modified
- `instances` is a dictionnary where the callback will add information needed to fill a specific form

FillRelayDomainInstances When a relay domain is being modified, this event is raised to fill extra forms.

Callback prototype:

```
def callback(user, rdomain, instances): pass
```

- `user` is the `User` instance corresponding to the currently logged in user
- `rdomain` is the `RelayDomain` instance being modified
- `instances` is a dictionnary where the callback will add information needed to fill a specific form

GetAnnouncement Some places in the interface let plugins add their own announcement (ie. message).

Callback prototype:

```
def callback(target): pass
```

- `target` is a string indicating the place where the announcement will appear:
- `loginpage` : corresponds to the login page

Callbacks listening to this event must return a list of string.

GetDomainActions Raised to request the list of actions available for the *domains* listing entry being displayed.

Callback prototype:

```
def callback(user, rdomain): pass
```

- `user` is the `User` instance corresponding to the currently logged in user
- `rdomain` is the `RelayDomain` instance being displayed

Must return a list of dictionaries, each dictionary containing at least the following entries:

```
{"name": "<action name>",
 "url": "<action url>",
 "title": "<action title>",
 "img": "<action icon>"}
```

GetDomainModifyLink Raised to request the modification url of the *domains* listing entry being displayed.

Callback prototype:

```
def callback(domain): pass
```

- `domain` is a model instance (`RelayDomain` for example)

Must return a dictionary containing at least the following entry:

```
{'url': '<modification url>'}
```

GetExtraLimitTemplates Raised to request extra limit templates. For example, the *postfix_relay_domains* extension define a template to limit the number of relay domains an administrator can create.

Callback prototype:

```
def callback(): pass
```

Must return a list of set. Each set must contain at least three entries:

```
[('<limit_name>', '<limit label>', '<limit help text>')]
```

GetExtraParameters Raised to request extra parameters for a given parameters form.

Callback prototype:

```
def callback(application, level): pass
```

- `application` is the name of the form's application (ie. admin, amavis, etc.)
- `level` is the form's level: `A` for admin or `U` for user

Must return a dictionary. Each entry must be a valid Django form field.

GetExtraRolePermissions Let extensions define new permissions for a given role.

Callback prototype:

```
def callback(rolename): pass
```

- `rolename` is the role's name (str)

Callbacks listening to this event must return a list of list. The second list level must contain exactly 3 strings: the application name, the model name and the permission name. Example:

```
[
    ["core", "user", "add_user"],
    ["core", "user", "change_user"],
    ["core", "user", "delete_user"],
]
```

GetExtraRoles Let extensions define new administrative roles (will be used to create or modify an account).

Callback prototype:

```
def callback(user, account): pass
```

- `user` is a `User` instance corresponding to the currently logged in user
- `account` is the account being modified (None on creation)

Callbacks listening to this event must return a list of 2uple (two strings) which respect the following format: `(value, label)`.

GetStaticContent Let extensions add static content (ie. CSS or javascript) to default pages. It is pretty useful for functionalities that don't need a template but need javascript stuff.

Callback prototype:

```
def callback(caller, st_type, user): pass
```

- `caller` is name of the application (or the location) responsible for the call
- `st_type` is the expected static content type (`css` or `js`)
- `user` is a `User` instance corresponding to the currently logged in user

Callbacks listening to this event must return a list of string.

ImportObject Raised to request the function handling an object being imported from CSV.

Callback prototype:

```
def callback(objtype): pass
```

`objtype` is the type of object being imported

Must return a list of function. A valid import function must respect the following prototype:

```
def import_function(user, row, formopts): pass
```

- `user` is the `User` instance corresponding to the currently logged in user
- `row` is a string containing the object's definition (CSV format)
- `formopts` is a dictionary that may contain options

InitialDataLoaded Raised a initial data has been loaded for a given extension.

Callback prototype:

```
 def callback(extname); pass

``extname`` is the extension name (str)
```

MailboxAliasCreated Raised when a new mailbox alias is created.

Callback prototype:

```
def callback(user, mailbox_alias): pass
```

- `user` is the new domain alias owner (`User` instance)
- `mailbox_alias` is the new mailbox alias (`Alias` instance)

MailboxAliasDeleted Raised when an existing mailbox alias is about to be deleted.

Callback prototype:

```
def callback(mailbox_alias): pass
```

- `mailbox_alias` is an `Alias` instance

MailboxCreated Raised when a new mailbox is created.

Callback prototype:

```
def callback(user, mailbox): pass
```

- `user` is the new mailbox's owner (`User` instance)
- `mailbox` is the new mailbox (`Mailbox` instance)

MailboxDeleted Raised when an existing mailbox is about to be deleted.

Callback prototype:

```
def callback(mailbox): pass
```

- `mailbox` is a `Mailbox` instance

MailboxModified Raised when an existing mailbox is modified.

Callback prototype:

```
def callback(mailbox): pass
```

- `mailbox` is the `Mailbox` modified instance. It contains a `old_full_address` extra field to check if the address was modified.

PasswordChange Raised just before a *password change* action.

Callback prototype:

```
def callback(user): pass
```

- `user` is a `User` instance

Callbacks listening to this event must return a list containing either `True` or `False`. If at least one `True` is returned, the *password change* will be cancelled (ie. changing the password for this user is disabled).

TopNotifications Lets extensions subscribe to the global notification service (located inside the top bar).

Callback prototype:

```
def callback(user, include_all): pass
```

- `request` is a `Request` instance
- `include_all` is a boolean indicating if empty notifications must be included into the result or not

Callbacks listening to this event must return a list of dictionary, each dictionary containing at least the following entries:

```
{"id": "<notification entry ID>",
 "url": "<associated URL>",
 "text": "<text to display>"}
```

If your notification needs a counter, you can specify it by adding the two following entries in the dictionary:

{“counter”: <associated counter>, “level”: “<info|success|warning|error>”}

UserLogin Raised when a user logs in.

Callback prototype:

```
def callback(request, username, password): pass
```

UserLogout Raised when a user logs out.

Callback prototype:

```
def callback(request): pass
```

UserMenuDisplay Raised when a user menu is about to be displayed.

Callback prototype:

```
def callback(target, user): pass
```

The `target` argument indicates which menu is being displayed. Possible values are:

- `options_menu`: corresponds to the top-right user menu
- `uprefs_menu`: corresponds to the menu bar available inside the *User preferences* page
- `top_menu`: corresponds to the top black bar

All the callbacks that listen to this event must return a list of dictionnaries (corresponding to menu entries). Each dictionnary must contain at least the following keys:

```
{"name" : "a_name_without_spaces",
 "label" : _("The menu label"),
 "url" : reverse("your_view")}
```

RelayDomainAliasCreated Raised when a new relay domain alias is created.

Callback prototype:

```
def callback(user, rdomain_alias): pass
```

- `user` is the new relay domain alias owner (`User` instance)
- `rdomain_alias` is the new relay domain alias (`DomainAlias` instance)

RelayDomainAliasDeleted Raised when an existing relay domain alias is about to be deleted.

Callback prototype:

```
def callback(rdomain_alias): pass
```

- `rdomain_alias` is a `RelayDomainAlias` instance

RelayDomainCreated Raised when a new relay domain is created.

Callback prototype:

```
def callback(user, rdomain): pass
```

- `user` corresponds to the `User` object creating the relay domain (its owner)
- `rdomain` is a `RelayDomain` instance

RelayDomainDeleted Raised when an existing relay domain is about to be deleted.

Callback prototype:

```
def callback(rdomain): pass
```

- `rdomain` is a `RelayDomain` instance

RelayDomainModified Raised when a relay domain has been modified.

Callback prototype:

```
def callback(rdomain): pass
```

- **`rdomain` is the modified `RelayDomain` instance, it contains an** extra `oldname` field which contains the old name

RoleChanged Raised when the role of an account is about to be changed.

Callback prototype:

```
def callback(account, role): pass
```

- `account` is the account being modified
- `role` is the new role (string)

SaveExtraFormFields Raised to save extra fields declared using *ExtraFormFields*.

Callback prototype:

```
def callback(form_name, instance, values): pass
```

- `form_name` is a string used to distinguish a specific form
- `instance` is a django model instance related to `form_name`
- `values` is a dictionary containing the form's values

UserCanSetRole Raised to check if a user is allowed to set a given role to an account.

Callback prototype:

```
def callback(user, role, account): pass
```

- `user` is the `User` instance corresponding to the currently logged in user
- `role` is the role `user` tries to set
- `account` is the account being modified (None on creation)

Must return a list containing `True` or `False` to indicate if this user can is allowed to set `role`.

2.6 Contributors

- Antidot
- Bearstech
- Dalnix