django mail admin Documentation

Release 0.1.1

Denis Bobrov

Mar 11, 2018

Contents

Contents:

CHAPTER 1

Django Mail Admin

The one and only django app to receive & send mail with templates and multiple configurations.

1.1 Features

- Everything django-mailbox has
- Everything django-post-office has
- Database configurations - activate an outbox to send from, activate a mailbox to receive from
- Templates
- Translatable
- Mailings - using send_many() or 'cc' and 'bcc' or even recipients - all of those accept comma-separated lists of emails

1.1.1 Dependencies

- django >= 1.9
- django-jsonfield

1.2 Documentation

The full documentation is at https://django_mail_admin.readthedocs.io.

1.3 Quickstart

Q: What versions of Django/Python are supported? **A**: Take a look at https://travis-ci.org/delneg/django_mail_admin

Install django mail admin:

```
pip install django_mail_admin
```

Add it to your *INSTALLED_APPS*:

```
INSTALLED_APPS = (
    ...
    'django_mail_admin',
    ...
)
```

- Run `migrate`:

  ```
  python manage.py migrate django_mail_admin
  ```

- Set `django_mail_admin.backends.CustomEmailBackend` as your `EMAIL_BACKEND` in django's `settings.py`:

  ```
  EMAIL_BACKEND = 'django_mail_admin.backends.CustomEmailBackend'
  ```

- Set cron/Celery/RQ job to send/receive email, e.g.

  ```
  * * * * * (cd $PROJECT; python manage.py send_queued_mail --processes=1 >>
  ↪$PROJECT/cron_mail.log 2>&1)
  * * * * * (cd $PROJECT; python manage.py get_new_mail >> $PROJECT/cron_mail_
  ↪receive.log 2>&1)
  0 1 * * * (cd $PROJECT; python manage.py cleanup_mail --days=30 >> $PROJECT/cron_
  ↪mail_cleanup.log 2>&1)
  ```

Note: Once you have entered a mailbox to receive emails, you can easily verify that you have properly configured your mailbox by either:

- From the Django Admin, using the 'Get New Mail' action from the action dropdown on the Mailbox changelist
- *Or* from a shell opened to your project's directory, using the `get_new_mail` management command by running:

  ```
  python manage.py get_new_mail
  ```

If you have also configured the Outbox, you can verify that it is working, e.g.

```
from django_mail_admin import mail, models

mail.send(
    'from@example.com',
    'recipient@example.com', # List of email addresses also accepted
    subject='My email',
    message='Hi there!',
    priority=models.PRIORITY.now,
    html_message='Hi <strong>there</strong>!',
)
```

1.4 Custom Email Backends

By default, `django_mail_admin` uses custom Email Backends that looks up for Outbox models in database. If you want to use a different backend, you can do so by configuring `BACKENDS`, though you will not be able to use Outboxes and will have to set EMAIL_HOST etc. in django's `settings.py`.

For example if you want to use django-ses:

```
DJANGO_MAIL_ADMIN = {
    'BACKENDS': {
        'default': 'django_mail_admin.backends.CustomEmailBackend',
        'smtp': 'django.core.mail.backends.smtp.EmailBackend',
        'ses': 'django_ses.SESBackend',
    }
}
```

You can then choose what backend you want to use when sending mail:

```
# If you omit `backend_alias` argument, `default` will be used
mail.send(
    'from@example.com',
    ['recipient@example.com'],
    subject='Hello',
)

# If you want to send using `ses` backend
mail.send(
    'from@example.com',
    ['recipient@example.com'],
    subject='Hello',
    backend='ses',
)
```

1.5 Optional requirements

1. *django_admin_row_actions* for some useful actions in the admin interface
2. *requests* & social_django for Gmail

1.6 FAQ

Q: Why did you write this?

A: In order to get both email sending & receiving you'll have to install post_office AND django_mailbox. Even if you do, you'll have to work on admin interface for it to look prettier, somehow link replies properly etc. So I've decided merging those two and clearing the mess in between them as well as adding some other useful features.

Q: Why did you remove support for Python 2?

A: Because f*ck python2. Really, it's been 9 (NINE!) years since it came out. Go ahead and check out https://github.com/brettcannon/caniusepython3

Q: Why is it named django_mail_admin, what does it have to do with admin ?

A: Well, the first version of this package (which was living just in a really large admin.py) was used for easy mail management using standard Django admin interface.

Q: What languages are available?

A: Currently there's Russian and English languages available. Feel free to add yours:

```
source <YOURVIRTUALENV>/bin/activate
python manage.py makemessages -l YOUR_LOCALE -i venv
python manage.py compilemessages -l YOUR_LOCALE
```

Q: Why did you delete support for multi-lingual templates?

A: Well, we have django-model-translations for that. You can easily fork this app and override EmailTemplate model (models/templates.py) accordingly. I think there's no need for such an overhead in a mail-related app.

Q: I don't want my outgoing emails to be queued for sending after saving them in the admin interface, what do i do?

A: Just override OutgoingEmailAdmin's save_model method.

Q: Can i get in touch with you? I want a new feature to be implemented/bug fixed!

A: Feel free to reach me out using issues and pull requests, I'll review them all and answer when I can.

1.7 Running Tests

Does the code actually work?

```
source <YOURVIRTUALENV>/bin/activate
(myenv) $ pip install tox
(myenv) $ tox
```

1.8 Credits

Tools used in rendering this package:

- Cookiecutter
- cookiecutter-djangopackage

CHAPTER 2

Usage

After you've installed Django Mail Admin:

Send a simple email is really easy:

```
from django_mail_admin import mail, models

mail.send(
    'from@example.com',
    'recipient@example.com', # List of email addresses also accepted
    subject='My email',
    message='Hi there!',
    priority=models.PRIORITY.now,
    html_message='Hi <strong>there</strong>!',
)
```

If you want to use templates: create an `EmailTemplate` instance via `admin` or `manually` and do the following:

```
from post_office import mail, models

template = models.EmailTemplate.objects.create(name='first', description='desc',␣
↪subject='{{id}}',
                                               email_html_text='{{id}}')
email = mail.create('from@example.com',
                    'recipient@example.com', # List of email addresses also accepted
                    template=template,
                    priority=models.PRIORITY.now)

models.TemplateVariable.objects.create(name='id', value=1, email=email)
models.OutgoingEmail.objects.get(id=email.id).dispatch() # re-get it from DB for␣
↪template variable to kick in, not needed when sending emails from queue via cron/
↪celery/etc.
# OR
mail.send('from@example.com',
          'recipient@example.com', # List of email addresses also accepted
           template=template,
```

```
            priority=models.PRIORITY.now,
            variable_dict={'id': 1})
```

2.1 Template variables & tags

`Django mail admin` supports Django's template tags and variables. It is important to note, however, that due to usage of TemplateVariable db-model, only strings can be stored as value, and anything in the template will be treated as a string.

For example, `'foo': [5,6]` when called in template as `{{ foo|first }}` will result in `[`, not in `5`. Please keep this in mind.

As an example of usage, if you put "Hello, {{ name }}" in the subject line and pass in `{'name': 'Alice'}` as variable_dict, you will get "Hello, Alice" as subject:

```
from post_office import mail, models

models.EmailTemplate.objects.create(
    name='morning_greeting',
    subject='Morning, {{ name|capfirst }}',
    content='Hi {{ name }}, how are you feeling today?',
    html_content='Hi <strong>{{ name }}</strong>, how are you feeling today?',
)

mail.send(
    'from@example.com',
    'recipient@example.com', # List of email addresses also accepted
    template=template,
    priority=models.PRIORITY.now,
    variable_dict={'name': 'alice'})
)

# This will create an email with the following content:
subject = 'Morning, Alice',
content = 'Hi alice, how are you feeling today?'
content = 'Hi <strong>alice</strong>, how are you feeling today?'
```

2.2 mail.send()

`mail.send` is the most important function in this library, it takes these arguments:

Argument	Required	Description
recipients	No	list of recipient email addresses
sender	Yes	Defaults to `settings.DEFAULT_FROM_EMAIL`, display name is allowed (`John <john@a.com>`)
subject	No	Email subject (if `template` is not specified)
message	No	Email content (if `template` is not specified)
html_message	No	HTML content (if `template` is not specified)
template	No	`EmailTemplate` instance
cc	No	list emails, will appear in `cc` field
bcc	No	list of emails, will appear in *bcc* field
attachments	No	Email attachments - A dictionary where the keys are the filenames and the values are either: • files • file-like objects • full path of the file
variables_dict	No	A dictionary, used to render templated email
headers	No	A dictionary of extra headers on the message
scheduled_time	No	A date/datetime object indicating when the email should be sent
priority	No	`high`, `medium`, `low` or `now` (send_immediately)
backend	No	Alias of the backend you want to use. `default` will be used if not specified.

Here are a few examples.

If you just want to send out emails without using database templates. You can call the `send` command without the `template` argument.

```
from django_mail_admin import mail

mail.send(
    'from@example.com',
    ['recipient1@example.com'],
    subject='Welcome!',
    message='Welcome home, {{ name }}!',
    html_message='Welcome home, <b>{{ name }}</b>!',
    headers={'Reply-to': 'reply@example.com'},
    scheduled_time=date(2019, 1, 1),
    variables_dict={'name': 'Alice'},
)
```

`django_mail_admin` is also task queue friendly. Passing `now` as priority into `send_mail` will deliver the email

right away (instead of queuing it), regardless of how many emails you have in your queue:

```
from django_mail_admin import mail, models

mail.send(
    'from@example.com',
    ['recipient1@example.com'],
    template=models.EmailTemplate.objects.get(name='welcome'),
    variables_dict={'foo': 'bar'},
    priority='now',
)
```

This is useful if you already use something like django-rq to send emails asynchronously and only need to store email related activities and logs.

If you want to send an email with attachments:

```
from django.core.files.base import ContentFile
 from django_mail_admin import mail, models

mail.send(
    ['recipient1@example.com'],
    'from@example.com',
    template=models.EmailTemplate.objects.get(name='welcome'),
    variables_dict={'foo': 'bar'},
    priority='now',
    attachments={
        'attachment1.doc': '/path/to/file/file1.doc',
        'attachment2.txt': ContentFile('file content'),
        'attachment3.txt': { 'file': ContentFile('file content'), 'mimetype': 'text/
↪plain'},
    }
)
```

2.3 send_many()

`send_many()` is much more performant (generates less database queries) when sending a large number of emails. `send_many()` is almost identical to `mail.send()`, with the exception that it accepts a list of keyword arguments that you'd usually pass into `mail.send()`:

```
from from django_mail_admin import mail

first_email = {
    'sender': 'from@example.com',
    'recipients': ['alice@example.com'],
    'subject': 'Hi!',
    'message': 'Hi Alice!'
}
second_email = {
    'sender': 'from@example.com',
    'recipients': ['bob@example.com'],
    'subject': 'Hi!',
    'message': 'Hi Bob!'
}
kwargs_list = [first_email, second_email]
```

```
mail.send_many(kwargs_list)
```

Attachments are not supported with `mail.send_many()`.

2.4 Management Commands

- `send_queued_mail` - send queued emails, those aren't successfully sent will be marked as `failed`. Accepts the following arguments:

Argument	Description
`--processes` or `-p`	Number of parallel processes to send email. Defaults to 1
`--lockfile` or `-L`	Full path to file used as lock file. Defaults to `/tmp/post_office.lock`

- `cleanup_mail` - delete all emails created before an X number of days (defaults to 90).

Argument	Description
`--days` or `-d`	Number of days to filter by.

- `get_new_mail` - receive new emails for all mailboxes or, if any args passed - filtered, e.g.:

```
python manage.py get_new_mail `test`
```

Set cron/Celery/RQ job to send/receive email, e.g.

```
* * * * * (cd $PROJECT; python manage.py send_queued_mail --processes=1 >> $PROJECT/
↪cron_mail.log 2>&1)
* * * * * (cd $PROJECT; python manage.py get_new_mail >> $PROJECT/cron_mail_receive.
↪log 2>&1)
0 1 * * * (cd $PROJECT; python manage.py cleanup_mail --days=30 >> $PROJECT/cron_mail_
↪cleanup.log 2>&1)
```

If you use uWSGI as application server, add this short snipped to the project's `wsgi.py` file:

```
from django.core.wsgi import get_wsgi_application

application = get_wsgi_application()

# add this block of code
try:
    import uwsgidecorators
    from django.core.management import call_command

    @uwsgidecorators.timer(10)
    def send_queued_mail(num):
        """Send queued mail every 10 seconds"""
        call_command('send_queued_mail', processes=1)

except ImportError:
    print("uwsgidecorators not found. Cron and timers are disabled")
```

Alternatively you can also use the decorator `@uwsgidecorators.cron(minute, hour, day, month, weekday)`. This will schedule a task at specific times. Use `-1` to signal any time, it corresponds to the uWSGI in cron.

Please note that `uwsgidecorators` are available only, if the application has been started with **uWSGI**. However, Django's internal `./manage.py runserver` also access this file, therefore wrap the block into an exception handler as shown above.

This configuration is very useful in environments, such as Docker containers, where you don't have a running cron-daemon.

2.5 Logging

You can configure Django Mail Admin's logging from Django's settings.py. For example:

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "django_mail_admin": {
            "format": "[%(levelname)s]%(asctime)s PID %(process)d: %(message)s",
            "datefmt": "%d-%m-%Y %H:%M:%S",
        },
    },
    "handlers": {
        "django_mail_admin": {
            "level": "DEBUG",
            "class": "logging.StreamHandler",
            "formatter": "django_mail_admin"
        },
        # If you use sentry for logging
        'sentry': {
            'level': 'ERROR',
            'class': 'raven.contrib.django.handlers.SentryHandler',
        },
    },
    'loggers': {
        "django_mail_admin": {
            "handlers": ["django_mail_admin", "sentry"],
            "level": "INFO"
        },
    },
}
```

2.6 Django Admin integration

Integration with Django Admin interface is provided. In there, you can send & receive emails, configure Outbox'es and Mailbox'es,

and if you've installed django-admin-row-actions you will have easy access to many features.

The admin interface integration will only be enabled if **DJANGO_MAILADMIN_ADMIN_ENABLED** setting is set to True (default is True).

CHAPTER 3

Settings

You should specify settings in your settings.py like this:

```
DJANGO_MAIL_ADMIN = {
        'BATCH_SIZE': 1000,
        'LOG_LEVEL': 1,
    }
```

Here's a list of available settings:

3.1 Settings for outgoing email

Setting	Default	Description
BATCH_SIZE	100	How many email's to send at a time. Used in *mail.py/get_queued*
THREADS_PER_PROCESS	3	How many threads to use when sending emails
DEFAULT_PRIORITY	'medium'	Priority, which is assigned to new email if not given specifically
LOG_LEVEL	2	Log level. 0 - log nothing, 1 - log errors, 2 - log errors and successors
SENDING_ORDER	['-priority']	Sending order for emails. If you want to send queued emails in FIFO order, set this to ['created']

3.2 Settings for incoming email

Setting	Default	Description
STRIP_UNALLOWED_MIMETYPES	False	Controls whether or not we remove mimetypes not specified in ALLOWED_MIMETYPES from the message prior to storage.
ALLOWED_MIMETYPES	['text/plain', 'text/html']	Has no effect if STRIP_UNALLOWED_MIMETYPES is set to False, otherwise - specifies allowed mimetypes that will not be stripped
TEXT_STORED_MIMETYPES	['text/plain', 'text/html']	A list of mimetypes that will remain stored in the text body of the message in the database.
ALTERED_MESSAGE_HEADER	'X-Django-Mail-Admin-Altered-Message'	Header to add to a message payload part in the event that the message cannot be reproduced accurately
ATTACHMENT_INTERPOLATION_HEADER	'X-Django-Mail-Admin-Interpolate-Attachment'	Header to add to the temporary 'dehydrated' message body in lieu of a non-text message payload component. The value of this header will be used to 'rehydrate' the message into a proper e-mail object in the event of a message instance's get_email_object method being called.
ATTACHMENT_UPLOAD_TO	'mail_admin_attachments/%Y/%m/%d/'	Attachments will be saved to this location. Specifies the upload_to setting for the attachment FileField. For more info, consult Django docs
STORE_ORIGINAL_MESSAGE	False	Controls whether or not we store original messages in eml field
COMPRESS_ORIGINAL_MESSAGE	False	Defines whether we compress the stored original message (.eml becomes .emg.gz)
ORIGINAL_MESSAGE_COMPRESSION	6	Defines the level of original message compression, if enabled
DEFAULT_CHARSET	'iso8859-1'	The charset that is used by default when decoding emails

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/delneg/django_mail_admin/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

4.1.4 Write Documentation

django mail admin could always use more documentation, whether as part of the official django mail admin docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/delneg/django_mail_admin/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *django_mail_admin* for local development.

1. Fork the *django_mail_admin* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django_mail_admin.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django_mail_admin
$ cd django_mail_admin/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 django_mail_admin tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/delneg/django_mail_admin/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_django_mail_admin
```

CHAPTER 5

Credits

5.1 Development Lead

- Denis Bobrov <delneg@yandex.ru>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.1.1 (2017-12-29)

- Added migrations.

6.2 0.1.0 (2017-12-29)

- First release on PyPI.