django-mailer2 Documentation

Release master

Tauber, Beaven & APSL

Feb 26, 2024

CONTENTS

1 How it works **3**

2 Advantages **5**

3 It's a fork! **7**
- 3.1 History . . . 7
- 3.2 Differences . . . 7

4 Credit **9**

5 Index **11**
- 5.1 Installation . . . 11
- 5.2 Mail Views . . . 12
- 5.3 Enqueue and send . . . 17
- 5.4 Storage backends . . . 18
- 5.5 Settings . . . 20
- 5.6 Contributing . . . 21
- 5.7 Changelog . . . 23

Django Yubin allows you to create, send and manage emails in your Django projects. It follows the 12-factors app methodology.

CHAPTER

ONE

HOW IT WORKS

For creating and composing emails, Yubin provides class-based views that use standard Django templates.

For sending and queuing emails, Yubin replaces the standard Django Email Backend with its own. Instead of sending emails synchronously trough a SMTP server, Yubin saves emails in your database (and optionally in a file storage) and sends them asynchronously using the Celery distributed task queue.

CHAPTER

TWO

ADVANTAGES

- Create and compose emails reusing your code easily with class-based views.
- Your app can respond requests faster because other process/worker is managing the connection with the SMTP server for sending emails.
- Scale out easily adding more Celery workers.
- Emails are saved in the database, you can see, manage and enqueue them from the Django Admin.
- Yubin provides settings to avoid sending emails during development.

CHAPTER

THREE

IT'S A FORK!

Yubin was forked from a django-mailer-2 fork which is a fork form Chris Beaven's fork of James Tauber's django-mailer.

3.1 History

Chris Beaven started a fork of django-mailer and it got to the point when it would be rather difficult to merge back. The fork was then renamed to the completely unimaginative "django mailer 2".

At APSL, we were always using this project together with mailviews, so we joined both together and started to add our own features.

3.2 Differences

Some of the larger differences in django yubin:

- It's integrated with django-mailviews classes.
- It saves a rendered version of the email, so HTML and other attachments are handled fine.
- This rendered emails can be saved in diferent sotrages: database, file system, AWS S3 or even your own custom storage.
- Models have been completely refactored for a better logical separation of data.
- Provides replacements for `send_mail`, `mail_admins` and `mail_managers` Django functions.
- Uses Celery distributed task queue instead of Cron.
- Task to remove old emails so the database does not increase so much.
- Improved admin configuration.
- Added a demo project for development and to see it in action in the admin.
- Support for runing tests without having to install and configure a Django application.
- Added CI and code coverage.
- Added a health check view.

CHAPTER

FOUR

CREDIT

At the time of the fork, the primary authors of django-mailer were James Tauber and Brian Rosner. Additional contributors include Michael Trier, Doug Napoleone and Jannis Leidel.

Original branch and the django-mailer-2 hard work comes from Chris Beaven.

django-mailviews from Disqus.

The name "Yubin" was suggested by @morenosan, he says it means "postal mail" in Japanesse, but who knows! :)

For Yubin contributors, have a look at Github's contributor's list and humans.txt.

CHAPTER

FIVE

INDEX

5.1 Installation

An obvious prerequisite of django-yubin is Django.

5.1.1 Installing Django Yubin

You can install the latest stable version from PyPI:

```
pip install django-yubin
```

or the latest commit from Github:

```
pip install -e git+http://github.com/APSL/django-yubin.git#egg=django-yubin
```

You can also download and install it manual with the `setup.py` installation script:

```
python setup.py install
```

5.1.2 Configuring your project

In your Django project's settings module, add `django_yubin` to your `INSTALLED_APPS` setting

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

Note that yubin doesn't queue all email by default, you must configure the email backend as

```
EMAIL_BACKEND = 'django_yubin.backends.QueuedEmailBackend'
MAILER_USE_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

More Yubin settings can be found in the *Settings sections*.

Add `yubin` urls in your main `urls.py` for using the health check

```
url(r'^yubin/', include('django_yubin.urls')),
```

Also, you need to setup Celery in your Django project and have at least one worker listening to the queue.

Finally, run database migrations

```
python manage.py migrate
```

With this setup emails will be saved entirely in the database. You can also configure Yubin to save emails in a different *storage*.

5.1.3 Upgrading from previous versions

Upgrading from versions < 0.1.8 to < 2.0.0

Version 0.1.8 added support for Django 1.9 and syncdb command no longer exists. If you are upgrading from a version < 0.1.8 and your models are already created you should execute

```
$ python manage.py migrate django_yubin --fake-initial
```

More details in https://docs.djangoproject.com/en/4.1/topics/migrations/#adding-migrations-to-apps

Upgrading from versions >= 0.1.8 to >= 2.0.0

Version 2.0.0 is a big reimplementation that uses Celery tasks instead of Cron jobs. This change needed considerable database schema changes but the database migrations take care of all. Keep in mind that:

- These database schema changes can not be undone. Once you migrate to version >= 2 you can not go backwards and use again a version < 2 unless you have a previous database backup.
- Stop cron jobs before doing the migration to avoid sending emails in an undetermined migration state.
- Have Celery setup and configuration ready but with no workers running. One of the migrations generates tasks to enqueue emails that were enqueued so they will be sent later.
- Once the migration finishes and everything is OK, start Celery workers so enqueued emails will be sent.

5.2 Mail Views

Rendering and sending emails in Django can quickly become repetitive and error-prone. By encapsulating message rendering within view classes, you can easily compose messages in a structured and clear manner.

The idea behind the method is identhical to the template class based view: you can select the template to use, in our case one for the subject and one for the body (and one extra for the html), you can pass the data you need overriding the `get_context_data` method and the message rendering is made in `render_to_message` where you can also customize the parameters as sender, cc, cco, etc. or delay the decision to the `send` method.

Managing mails could become a crazy thing quite fast, so the idea is to be able to organize the mail templates in folders and use the mailview classes to provide them with the data you need.

Using inheritance in templates, mixins and inheritance will give you again the control.

5.2.1 Your first email

Let's suppose we want to send a notificantion message to a mailing list. We don't have a customized email, but we want to be able to render the e-mail

```
import datetime
from django_yubin.message_views import TemplatedEmailMessageView

class NewsletterView(TemplatedEmailMessageView):
    subject_template_name = 'emails/newsletter/subject.txt'
    body_template_name = 'emails/newsletter/body.txt'

    def get_context_data(self, **kwargs):
        """
        here we can get the addtional data we want
        """
        context = super().get_context_data(**kwargs)
        context['day'] = datetime.date.today()
        return context

# Instantiate and send a message.
NewsletterView().send(to=('mynewsletter@example.com', ))
```

This would render and send a message to the newsletter with the `DEFAULT_FROM_EMAIL` emails settings. Sometimes we'd like to send it with different e-mail, so we can customize it as

```
NewsletterView().send(from_email='no-reply@example.com',
                      to=('mynewsletter@exemple.com', ))
```

Any keywords you pass in send will be forwarded to the django mail calss, so you can use the same parameters you have in the Django `EmailMessage` class documentation:

- **from_email**: The sender's address. Both fred@example.com and Fred <fred@example.com> forms are legal. If omitted, the DEFAULT_FROM_EMAIL setting is used.
- **to**: A list or tuple of recipient addresses.
- **bcc**: A list or tuple of addresses used in the "Bcc" header when sending the email.
- **cc**: A list or tuple of recipient addresses used in the "Cc" header when sending the email.

Instead of using `send` you can use `render_to_message` method. Its parameters are the same as the `send` method, but instead of sending the e-mail it will return you an instance of `EmailMessage` that you can use to customize the e-mail before sending it.

In our example, we could write:

```
import datetime
from django_yubin.message_views import TemplatedEmailMessageView

class NewsletterView(TemplatedEmailMessageView):
    subject_template_name = 'emails/newsletter/subject.txt'
    body_template_name = 'emails/newsletter/body.txt'

    def render_to_message(self, extra_context=None, **kwargs):
        kwargs['to'] = ('mynewsletter@example.com',)
        kwargs['from_email'] = 'no-reply@example.com'
```

(continues on next page)

(continued from previous page)

```
        return super().render_to_message(extra_context, **kwargs)

    def get_context_data(self, **kwargs):
        """
        here we can get the addtional data we want
        """
        context = super().get_context_data(**kwargs)
        context['day'] = datetime.date.today()
        return context

# Instantiate and send a message.
NewsletterView().send()
```

Supose now that we wan't to send a second newsletter, the monthly one for example, then we could just write

```
class MonthlyNewsletterView(NewsletterView):
    subject_template_name = 'emails/newsletter/monthly_subject.txt'
    body_template_name = 'emails/newsletter/monthly_body.txt'

MonthlyNewsletterView().send()
```

5.2.2 HTML emails

In the previous example we have sent just text emails. If we want to send HTML email we need also an additional template to render the HTML content. You just have to inherit your class from `TemplatedHTMLEmailMessageView` and write the template you're going to use in `html_body_template_name`, so usually we'll have something like

```
import datetime
from django_yubin.message_views import TemplatedHTMLEmailMessageView

class NewsletterView(TemplatedHTMLEmailMessageView):
    subject_template_name = 'emails/newsletter/subject.txt'
    body_template_name = 'emails/newsletter/body.txt'
    html_body_template_name = 'emails/newsletter/body_html.html'

    def render_to_message(self, extra_context=None, **kwargs):
        kwargs['to'] = ('mynewsletter@example.com',)
        kwargs['from_email'] = 'no-reply@example.com'
        return super().render_to_message(extra_context=None, **kwargs)

    def get_context_data(self, **kwargs):
        """
        here we can get the addtional data we want
        """
        context = super().get_context_data(**kwargs)
        context['day'] = datetime.date.today()
        return context

# Instantiate and send a message.
NewsletterView().send()
```

Usually, in HTML emails you need to link files from your site. *MEDIA_URL* and *STATIC_URL* variables are available

in the template context. These variables are full urls so you need to have *django.contrib.sites* and *SITE_ID* properly set in your *SETTINGS.py*.

5.2.3 Attachments

To add an attachment to your mail you have to remember that *render_to_message* returns a *EmailMessage* instance, so you can use https://docs.djangoproject.com/en/dev/topics/email/#emailmessage-objects.

As usually we send just an attachment, we have created a class for that just passing the file name or a file object: `TemplatedAttachmentEmailMessageView`. For example, if we want to send in our newsletter a pdf file we could do

```
import datetime
from django_yubin.message_views import TemplatedAttachmentEmailMessageView

class NewsletterView(TemplatedAttachmentEmailMessageView):
    subject_template_name = 'emails/newsletter/subject.txt'
    body_template_name = 'emails/newsletter/body.txt'
    html_body_template_name = 'emails/newsletter/body_html.html'

    def render_to_message(self, extra_context=None, **kwargs):
        kwargs['to'] = ('mynewsletter@example.com',)
        kwargs['from_email'] = 'no-reply@example.com'
        return super().render_to_message(extra_context=None, **kwargs)

    def get_context_data(self, **kwargs):
        """
        here we can get the addtional data we want
        """
        context = super().get_context_data(**kwargs)
        context['day'] = datetime.date.today()
        return context

# Instantiate and send a message.
attachment = os.path.join(OUR_ROOT_FILES_PATH, 'newsletter/attachment.pdf')
NewsletterView().send(attachment=attachment, mimetype="application/pdf")
```

As an attachment you must provide the full file path or the data stream.

5.2.4 Multiple attachments

Sending multiple attachments works the same way but using the class `TemplatedMultipleAttachmentsEmailMessageView`. Example:

```
import datetime
from django_yubin.message_views import TemplatedMultipleAttachmentsEmailMessageView

class NewsletterView(TemplatedMultipleAttachmentsEmailMessageView):
    subject_template_name = 'emails/newsletter/subject.txt'
    body_template_name = 'emails/newsletter/body.txt'
    html_body_template_name = 'emails/newsletter/body_html.html'
```

(continues on next page)

(continued from previous page)

```
    def render_to_message(self, extra_context=None, **kwargs):
        kwargs['to'] = ('mynewsletter@example.com',)
        kwargs['from_email'] = 'no-reply@example.com'
        return super().render_to_message(extra_context=None, **kwargs)

    def get_context_data(self, **kwargs):
        """
        here we can get the addtional data we want
        """
        context = super().get_context_data(**kwargs)
        context['day'] = datetime.date.today()
        return context

# Instantiate and send a message.
attachments = [
    {"attachment": os.path.join(OUR_ROOT_FILES_PATH, 'newsletter/attachment.pdf'),
↪"filename": "attachment.pdf"},
    {"attachment": os.path.join(OUR_ROOT_FILES_PATH, 'newsletter/attachment2.pdf'),
↪"filename": "attachment2.pdf"},
    {"attachment": os.path.join(OUR_ROOT_FILES_PATH, 'newsletter/attachment3.pdf'),
↪"filename": "attachment3.pdf"},
]
NewsletterView().send(attachments=attachments)
```

5.2.5 Email to a user

The `send` method can receive any extra context that you need to create your emails. Even it can be usefull as a quick shortcut, it's not e good pattern

```
from django_yubin.message_views import TemplatedEmailMessageView

# Subclass the TemplatedEmailMessageView adding the templates you want to render.
class WelcomeMessageView(TemplatedEmailMessageView):
    subject_template_name = 'emails/welcome/subject.txt'
    body_template_name = 'emails/welcome/body.txt'

# Instantiate and send a message.
WelcomeMessageView().send(extra_context={'user': user}, to=(user.email, ))
```

A better approach is to subclass `TemplatedEmailMessageView`. Its constructor accepts all the paameters that you need to generate the context and send the message. Example:

```
from django_yubin.message_views import TemplatedEmailMessageView

class WelcomeMessageView(TemplatedEmailMessageView):
    subject_template_name = 'emails/welcome/subject.txt'
    body_template_name = 'emails/welcome/body.txt'

    def __init__(self, user, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.user = user
```

(continues on next page)

(continued from previous page)

```
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['user'] = self.user
        return context

    def render_to_message(self, *args, **kwargs):
        assert 'to' not in kwargs  # this should only be sent to the user
        kwargs['to'] = (self.user.email, )
        return super().render_to_message(*args, **kwargs)

# Instantiate and send a message.
WelcomeMessageView(user).send()
```

In fact, you might find it helpful to encapsulate the above "message for a user" pattern into a mixin or subclass that provides a standard abstraction for all user-related emails.

5.3 Enqueue and send

5.3.1 Putting emails in the queue

Yubin replaces the standard Django Email Backend with its own. Instead of sending emails synchronously trough a SMTP server, Yubin saves and equeues emails in your database and creates Celery tasks to send them asynchronously using "the real" Django Email Backend.

```
# settings.py

# Yubin email backend that enqueue emails
EMAIL_BACKEND = 'django_yubin.backends.QueuedEmailBackend'

# "The real" email backend for sending emails
MAILER_USE_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

Now, you can call `send_mail` like you normally do in Django or using *mail views*

```
send_mail(subject, message_body, settings.DEFAULT_FROM_EMAIL, recipients)
# ...
WelcomeMessageView(user).send()
```

5.3.2 Tasks

Once you have your emails queued in the database, you can send, retry or delete them using the following Celery tasks:

- **send_email(message_pk)** Sends the email from the database with the given primay key.
- **retry_emails(max_retries=3)** Retry sending retryable emails (failed, blacklisted or discarded) enqueueing them again.
- **delete_old_emails(days=90)** Delete emails created before *days* days.

You don't usually need to create a `send_email` task, Yubin email backend does it automatically. For `retry_emails` and `delete_old_emails`, you can use Celery Beat to schedule periodic task.

Remember to have at least one Celery worker listening for tasks.

5.3.3 Commands

In addition to tasks, Yubin also provides a couple of commands to facilitate the development:

- **send_test_mail** Sends a single HTML email. Ideal for checking connection parameters.
- **create_email** Creates fake mails for testing unicode, emojis and attachments.
- **db2file** and **file2db** migrate emails between storage backends. Look at the *Storage backends* section for more details.

Execute `python manage.py THE_COMMAND --help` to see optional arguments.

5.3.4 Health check

If you have added `url(r'^yubin/', include('django_yubin.urls'))` to your `urls.py`, you can go to `http://localhost:8000/yubin/health` and see the health output result.

Response when there are no problems: HTTP 200

```
oldest_queued_email: 1 mins
emails_queued_too_old: no
threshold: 30 mins
```

Response if there are emails that have been too long enqueued: HTTP 500

```
# HTTP 500
oldest_queued_email: 45 mins
emails_queued_too_old: yes
threshold: 30 mins
```

You can parse this view's response in your check system and check the status code of the response.

Additionally, you can use a different threshold changing `settings.MAILER_HC_QUEUED_LIMIT_OLD` or passing a GET parameter *t*: http://localhost:8000/yubin/health?t=60

5.4 Storage backends

Yubin uses the database to save data about emails: dates, status (queued, sent…) and some fields to easy access, filtering and searching without parsing the email.

In addition, it needs to save full email contents somewhere. Storage backends are responsible of saving and retrieving these emails. You can choose one using the `settings.MAILER_STORAGE_BACKEND` variable.

5.4.1 DatabaseStorageBackend

`django_yubin.storage_backends.DatabaseStorageBackend`

By default, Yubin saves emails in the database. This is a simple solution that works well if you send few emails, they are text-only or you don't attach heavy files.

5.4.2 FileStorageBackend

`django_yubin.storage_backends.FileStorageBackend`

If that's not the case, the database can grow a lot with data that is read only a few times, it can increase the size of your backup, etc.

`FileStorageBackend` uses Django's default file storage to save emails in the file system and only the path in the database.

With the help of Django Storages you can also save emails in many other file/object storages: AWS S3, MinIO, Azure Storage, Google Cloud Storage, Dropbox, SFTPs, etc.

5.4.3 Custom storage backends

If you have different needs, you can also write your own custom storage backend:

- Inherit from the base class `django_yubin.storage_backends.BaseStorageBackend` and implement its abstract methods for getting and settings the email.
- You can have a look at the other storage backends and the comments in `django_yubin.models.Message` to have an idea.
- Set `settings.MAILER_STORAGE_BACKEND` with the path of your custom storage backend. For example, `"my_project.storage_backends.MyCustomStorageBackend"`.

You can also inherit from other already implemented storage backends. For example, let's write a custom storage backend that uses an inmutable file system storage:

```
# my_project.storage_backends.py

from django.core.files.storage import FileSystemStorage
from django_yubin.storage_backends import FileStorageBackend


class _InmutableFileSystemStorage(FileSystemStorage):
    def delete(self, name):
        pass


class InmutableFileStorageBackend(FileStorageBackend):
    storage = _InmutableFileSystemStorage()
```

```
# settings.py

MAILER_STORAGE_BACKEND = 'my_project.storage_backends.InmutableFileStorageBackend'
```

5.4.4 Migrate between storage backends

Yubin versions < 2.0.0 don't have storage backends and have all the emails saved in the database. When you upgrade Yubin to a version >= 4.0.0, by default it uses the `DatabaseStorageBackend` so you still have the emails in the database.

In case you want to use the `FileStorageBackend`, Yubin provides a Django command to do the migration moving emails from the database to the file storage. There is also the opposite command if you want to use again the `DatabaseStorageBackend`.

Warning: Consider to do a database and/or file storage backup before executing any of these commads.

DB2File

- Stop any cron, Celery worker, etc. that can send emails.
- Change `settings.MAILER_STORAGE_BACKEND` to `django_yubin.storage_backends.FileStorageBackend`.
- Execute:

```
python manage.py db2file
```

- Enable again your Celery workers.

File2DB

- Stop any cron, Celery worker, etc. that can send emails.
- Change `settings.MAILER_STORAGE_BACKEND` to `django_yubin.storage_backends.DatabaseStorageBackend`.
- Execute:

```
python manage.py file2db
```

- Or if you also want to delete the files in the file storage after saving them into the database, execute:

```
python manage.py file2db --delete
```

- Enable again your Celery workers.

If you use a different or custom storage backend and you want to migrate emails, you should write your own migration commad.

5.5 Settings

List of settings that can be added to your Django project. All are optional and have sane defaults.

MAILER_PAUSE_SEND

Provides a way to temporarily pause sending mails. Default is `False`.

If `True`, mail will be discarded and not be sent by any function.

MAILER_USE_BACKEND

The mail backend to use when actually sending emails. Default is `'django.core.mail.backends.smtp.EmailBackend'`.

Tip: You can use Django's console or dummy backends during development to avoid sending fake emails.

MAILER_TEST_MODE

When `True`, recipient addresses of all messages are replaced with the value of `MAILER_TEST_EMAIL` before being created and enqueued. An additional header `X-Yubin-Test-Original` will be added with the content of the original recipient addresses. Both `MAILER_TEST_MODE` and `MAILER_TEST_EMAIL` must evaluate to `True` to activate this mode. Default is `False`.

MAILER_TEST_EMAIL

Email address where all mail is sent when `MAILER_TEST_MODE` is `True`. Default is `''`.

MAILER_HC_QUEUED_LIMIT_OLD

If there are mails created or enqueued or in progress for more than `MAILER_HC_QUEUED_LIMIT_OLD` minutes, the HealthCheck view will show an error. Default is `30` minutes.

MAILER_STORAGE_BACKEND

Storage to save full emails. Default is `django_yubin.storage_backends.DatabaseStorageBackend`. You can also use `django_yubin.storage_backends.FileStorageBackend` or provide your own.

MAILER_STORAGE_DELETE

When deleting an email from the database, also delete its data from the storage. Default is `True`.

MAILER_FILE_STORAGE_DIR

Subdirectory to save emails when using the `FileStorageBackend`. Default is `yubin`.

5.6 Contributing

django-yubin is an open source project and improvements and bug reports are very appreciated.

You can contribute in many ways:

- Filling a bug on github.
- Creating a patch and sending the pull request.
- Help on testing and documenting.

When sending a pull request, please be sure that all tests and builds passes. In the next section you'll find information about how to write and run tests.

Please, follow the PEP8 coventions and in case you write additional features don't forget to write tests for them.

At APSL we use yubin for most of our own projects, so we'll try to mantain it as bug free and stable as possible. That said, we can't not guarantee that we could patch the program in the way you like, add that new feature, etc.

5.6.1 Demo Project

A demo/development project is provided in the `demo` directory. It is usefull during development for manual tests, checks and the admin site.

The project has a Docker Compose file with all the development infrastructure needed: a Postgres database, a Redis server for Celery tasks and a SMTP server that prints emails in the terminal.

Also, you will need Pipenv to manage Python dependencies and virtualenv.

Usage

- Create virtualenv and install dependencies

```
$ cd demo
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install -r requirements.txt
```

- Start servers

```
$ docker compose up -d
```

- The first time you need to run database migrations and create a superuser

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

- Start the development server

```
$ python manage.py runserver
```

- In another terminal, start a Celery worker and a Celery Beat scheduler

```
$ celery -A demo.celery worker -l info -B
```

Now you can open http://localhost:8000 in your browser and see the demo project running.

Once yo have finished, to stop everything you can:

- Press Ctrl+C to stop the `runserver` and the Celery worker.
- Run *$ docker compose down* to stop all the servers.

5.6.2 Tests

To run the tests you need a Python environment with the test dependencies. The easiest way is to use the same virtualenv that you have created for the demo project, it already has the test dependencies.

Once you are in the demo virtual environment, you can go to the root directory and run tests from there. Let's see a couple of examples:

```
# Run all tests with you current python environment
$ ./runtests.py

# Run specific tests with you current python environment
$ ./runtests.py tests.tests.test_backend
```

(continues on next page)

(continued from previous page)

```
# Use tox to run all tests with all available Python environments and see a
# code coverage report
$ tox

# The same but only for a specific subset of tests
$ tox -- tests.tests.test_backend
```

5.6.3 CI/CD

Continuous integration and deployment are done using Github Actions. Right now it runs tests and code coverage with Tox in PRs and pushes to *master* branch.

Please, be sure that everything is green before sending PRs.

Feel free to add yourself to `humans.txt` file in your PR.

5.6.4 Documentation

This documentation is built with Sphinx and is available at Read the Docs.

5.7 Changelog

This project adheres to Semantic Versioning.

Starting from version 2.0.0, the format is based on Keep a Changelog.

5.7.1 [2.0.3] - 2024-01-09

Fixed

- Fix race condition between Celery and database transactions (https://github.com/APSL/django-yubin/pull/74)

5.7.2 [2.0.2] - 2023-10-30

Changed

- Improve performance of data migration when migrating from versions < 2.0 (https://github.com/APSL/django-yubin/pull/69)

Fixed

- Perform unfolding of headers when parsing messages (https://github.com/APSL/django-yubin/pull/71)

5.7.3 [2.0.1] - 2023-08-10

Changed

- Ensure parsed e-mail message doesn't discard information (https://github.com/APSL/django-yubin/pull/67)

5.7.4 [2.0.0] - 2023-06-29

Changed

- Send and queue emails with Celery instead of with Cron.
- Drop priority headers (useless with queues).
- Storage backends to save emails in databases, file storages, etc.
- Cc and Bcc support.
- Supported versions: Python 3.8~3.11, Django 3.2~4.2, Celery 5.0~5.2.
- Migrate CI/CD from Travis to Github Actions.
- Docker Compose for external dependencies in development environment.
- Get django_yubin version programmatically.
- Update docs.

5.7.5 Older versions - 2022-01-17

- 1.7.1 - Remove abandoned `pyzmail36` dependency with `mail-parser`.
- 1.7.0 - Add optional `MAILER_MESSAGE_SEARCH_FIELDS` setting. It's a tuple of strings with the fields to use in `admin.Message.search_fields` attribute.
- 1.6.0 - Support for Django 3.0
- 1.5.0 - New TemplatedMultipleAttachmentsEmailMessageView to allow to send emails with more than 1 attachment.
- 1.4.1 - Detecting if messages are encoding using different encoding headers to be able to preview them (now base64, quoted-printable).
- 1.4.0 - Option added in status_mail command to return the output in json format.
- 1.3.1 - Fix unicode and encode errors: sending queued and non queued emails and in admin detail view.
- 1.3.0 - Allow to send emails inmediatly without being saved in database (priority «now-not-queued»). Add support for Python 3.7 and Django 2.1. Remove old code for Django < 1.3.
- 1.2.0 - Fix is_base64 detection. Add a «send_test_email» command to check connection parameters. New health check view. Don't open a connection if there are no messages in queue to send. Add a "date_sent" field to detect when the mail was sent.

- 1.1.0 - Fix attachment headers in TemplateAttachmentEmailMessagView making both "attachment" and "filename" args mandatory.
- 1.0.5 - Add missing paths in MANIFEST.in.
- 1.0.4 - Fix attachment visualization in the admin. Attach pdf in create_mail command. Solved Content-Transfer-Encoding issue.
- 1.0.3 - Fixed issue decoding the message payload, added support for django 1.9, updated changelog and added support to deploy the package from travis.
- 1.0.0 - Add support for Django 2.0 and remove django 1.8.
- 0.8.2 - Fix date created column in QueuedMessages admin.
- 0.8.1 - Ensure that LOCK_WAIT_TIMEOUT is never negative to avoid a bug in lockfile in systems which use a LinkFileLock.
- 0.8.0 - Use settings.MAILER_PAUSE_SEND to skip smtp connections. Fix UTF-8 encoding in messages. Fix encoding errors in email visualization in the admin.
- 0.7.0 - Fix template context bug for Django 1.11. Add Python 3.6 to CI and drop Python 3.3 and Django 1.9.
- 0.6.0 - Support for Python 3.6.
- 0.5.0 - Limit nº of emails sent by send_mail command. Update the debug handlers options for verbosity to accept v3.
- 0.4.0 - Support Django 1.11: subject and body are no longer unscaped, you need to add {% autoescape off %} to your non HTML templates.
- 0.3.1 - Delete unused template that caused an error with django-compressor offline. testmail command now generates HTML emails.
- 0.3.0 - Support Django >= 1.8 and <=1.10, Python 2.7, 3.3, 3.4 and 3.5. Re-send mails admin action. Fix bug in status_mail command. Demo project configured to send mails with the same mail fake-server used for tests.
- 0.2.3 - Removed {% load url from future %} to support Django 1.9. Now Django < 1.5 is not supported.
- 0.2.2 - Include migrations directory in .tar.gz in PyPi.
- 0.2.1 - Updated links to CI and Code Coverage Services
- 0.2.0 - Merged sergei-maertens contribution.
- 0.1.8 - Added migrations for Django 1.9 compatibility. See http://django-yubin.readthedocs.org/en/latest/install.html#upgrading-from-previous-versions
- 0.1.7 - Support for Django 1.8.
- 0.1.6 - Bugfixes.
- 0.1.5 - Bugfixes.
- 0.1.4 - Updated README.
- 0.1.3 - Fixed Python3 compatibility, thanks Marc, Cesc & Dani.
- 0.1.2 - Fixed Templates.
- 0.1.1 - Updated documentation and unit tests.