flask-mail Documentation

Release 0.9.1

Dan Jacob

Mar 26, 2019

Contents

One of the most basic functions in a web application is the ability to send emails to your users.

The **Flask-Mail** extension provides a simple interface to set up SMTP with your Flask application and to send messages from your views and scripts.

CHAPTER 1

Links

- documentation
- source
- changelog

CHAPTER 2

Installing Flask-Mail

Install with **pip** and **easy_install**:

```
pip install Flask-Mail
```

or download the latest version from version control:

```
git clone https://github.com/mattupstate/flask-mail.git
cd flask-mail
python setup.py install
```

If you are using **virtualenv**, it is assumed that you are installing flask-mail in the same virtualenv as your Flask application(s).

CHAPTER 3

Configuring Flask-Mail

Flask-Mail is configured through the standard Flask config API. These are the available options (each is explained later in the documentation):

- **MAIL_SERVER** : default **'localhost'**
- **MAIL_PORT** : default **25**
- **MAIL_USE_TLS** : default **False**
- **MAIL_USE_SSL** : default **False**
- **MAIL_DEBUG** : default **app.debug**
- **MAIL_USERNAME** : default **None**
- **MAIL_PASSWORD** : default **None**
- **MAIL_DEFAULT_SENDER** : default **None**
- **MAIL_MAX_EMAILS** : default **None**
- **MAIL_SUPPRESS_SEND** : default **app.testing**
- **MAIL_ASCII_ATTACHMENTS** : default **False**

In addition the standard Flask `TESTING` configuration option is used by **Flask-Mail** in unit tests (see below).

Emails are managed through a `Mail` instance:

```
from flask import Flask
from flask_mail import Mail

app = Flask(__name__)
mail = Mail(app)
```

In this case all emails are sent using the configuration values of the application that was passed to the `Mail` class constructor.

Alternatively you can set up your `Mail` instance later at configuration time, using the **init_app** method:

```
mail = Mail()

app = Flask(__name__)
mail.init_app(app)
```

In this case emails will be sent using the configuration values from Flask's `current_app` context global. This is useful if you have multiple applications running in the same process but with different configuration options.

CHAPTER 4

Sending messages

To send a message first create a `Message` instance:

```
from flask_mail import Message

@app.route("/")
def index():

    msg = Message("Hello",
                  sender="from@example.com",
                  recipients=["to@example.com"])
```

You can set the recipient emails immediately, or individually:

```
msg.recipients = ["you@example.com"]
msg.add_recipient("somebodyelse@example.com")
```

If you have set `MAIL_DEFAULT_SENDER` you don't need to set the message sender explicity, as it will use this configuration value by default:

```
msg = Message("Hello",
              recipients=["to@example.com"])
```

If the `sender` is a two-element tuple, this will be split into name and address:

```
msg = Message("Hello",
              sender=("Me", "me@example.com"))

assert msg.sender == "Me <me@example.com>"
```

The message can contain a body and/or HTML:

```
msg.body = "testing"
msg.html = "<b>testing</b>"
```

Finally, to send the message, you use the `Mail` instance configured with your Flask application:

```
mail.send(msg)
```

CHAPTER 5

Bulk emails

Usually in a web application you will be sending one or two emails per request. In certain situations you might want to be able to send perhaps dozens or hundreds of emails in a single batch - probably in an external process such as a command-line script or cronjob.

In that case you do things slightly differently:

```
with mail.connect() as conn:
    for user in users:
        message = '...'
        subject = "hello, %s" % user.name
        msg = Message(recipients=[user.email],
                      body=message,
                      subject=subject)

        conn.send(msg)
```

The connection to your email host is kept alive and closed automatically once all the messages have been sent.

Some mail servers set a limit on the number of emails sent in a single connection. You can set the max amount of emails to send before reconnecting by specifying the **MAIL_MAX_EMAILS** setting.

CHAPTER 6

Attachments

Adding attachments is straightforward:

```
with app.open_resource("image.png") as fp:
    msg.attach("image.png", "image/png", fp.read())
```

See the *API* for details.

If `MAIL_ASCII_ATTACHMENTS` is set to **True**, filenames will be converted to an ASCII equivalent. This can be useful when using a mail relay that modify mail content and mess up Content-Disposition specification when filenames are UTF-8 encoded. The conversion to ASCII is a basic removal of non-ASCII characters. It should be fine for any unicode character that can be decomposed by NFKD into one or more ASCII characters. If you need romanization/transliteration (i.e $\beta \rightarrow ss$) then your application should do it and pass a proper ASCII string.

CHAPTER 7

Unit tests and suppressing emails

When you are sending messages inside of unit tests, or in a development environment, it's useful to be able to suppress email sending.

If the setting `TESTING` is set to `True`, emails will be suppressed. Calling `send()` on your messages will not result in any messages being actually sent.

Alternatively outside a testing environment you can set `MAIL_SUPPRESS_SEND` to **False**. This will have the same effect.

However, it's still useful to keep track of emails that would have been sent when you are writing unit tests.

In order to keep track of dispatched emails, use the `record_messages` method:

```
with mail.record_messages() as outbox:

    mail.send_message(subject='testing',
                      body='test',
                      recipients=emails)

    assert len(outbox) == 1
    assert outbox[0].subject == "testing"
```

The **outbox** is a list of `Message` instances sent.

The blinker package must be installed for this method to work.

Note that the older way of doing things, appending the **outbox** to the `g` object, is now deprecated.

CHAPTER 8

Header injection

To prevent header injection attempts to send a message with newlines in the subject, sender or recipient addresses will result in a `BadHeaderError`.

CHAPTER 9

Signalling support

New in version 0.4.

Flask-Mail now provides signalling support through a `email_dispatched` signal. This is sent whenever an email is dispatched (even if the email is not actually sent, i.e. in a testing environment).

A function connecting to the `email_dispatched` signal takes a `Message` instance as a first argument, and the Flask app instance as an optional argument:

```
def log_message(message, app):
    app.logger.debug(message.subject)

email_dispatched.connect(log_message)
```

CHAPTER 10

API

Python Module Index

f

Index

F