Anymail Documentation

Release 7.0.0

Anymail contributors (see AUTHORS.txt)

Sep 07, 2019

Using Anymail

Version 7.0.0

Anymail integrates several transactional email service providers (ESPs) into Django, with a consistent API that lets you use ESP-added features without locking your code to a particular ESP.

It currently fully supports **Amazon SES, Mailgun, Mailjet, Postmark, SendinBlue, SendGrid,** and **SparkPost,** and has limited support for **Mandrill.**

Anymail normalizes ESP functionality so it "just works" with Django's built-in `django.core.mail` package. It includes:

- Support for HTML, attachments, extra headers, and other features of Django's built-in email
- Extensions that make it easy to use extra ESP functionality, like tags, metadata, and tracking, with code that's portable between ESPs
- Simplified inline images for HTML email
- Normalized sent-message status and tracking notification, by connecting your ESP's webhooks to Django signals
- "Batch transactional" sends using your ESP's merge and template features
- Inbound message support, to receive email through your ESP's webhooks, with simplified, portable access to attachments and other inbound content

Anymail is released under the BSD license. It is extensively tested against Django 1.11–2.2 (including Python 2.7, Python 3 and PyPy). Anymail releases follow semantic versioning.

CHAPTER 1

Documentation

1.1 Anymail 1-2-3

Here's how to send a message. This example uses Mailgun, but you can substitute Mailjet or Postmark or SendGrid or SparkPost or any other supported ESP where you see "mailgun":

1. Install Anymail from PyPI:

```
$ pip install django-anymail[mailgun]
```

   (The `[mailgun]` part installs any additional packages needed for that ESP. Mailgun doesn't have any, but some other ESPs do.)

2. Edit your project's `settings.py`:

```
INSTALLED_APPS = [
    # ...
    "anymail",
    # ...
]

ANYMAIL = {
    # (exact settings here depend on your ESP...)
    "MAILGUN_API_KEY": "<your Mailgun key>",
    "MAILGUN_SENDER_DOMAIN": 'mg.example.com',  # your Mailgun domain, if needed
}
EMAIL_BACKEND = "anymail.backends.mailgun.EmailBackend"  # or sendgrid.
↪EmailBackend, or...
DEFAULT_FROM_EMAIL = "you@example.com"  # if you don't already have this in␣
↪settings
SERVER_EMAIL = "your-server@example.com"  # ditto (default from-email for Django␣
↪errors)
```

3. Now the regular Django email functions will send through your chosen ESP:

```
from django.core.mail import send_mail

send_mail("It works!", "This will get sent through Mailgun",
          "Anymail Sender <from@example.com>", ["to@example.com"])
```

You could send an HTML message, complete with an inline image, custom tags and metadata:

```
from django.core.mail import EmailMultiAlternatives
from anymail.message import attach_inline_image_file

msg = EmailMultiAlternatives(
    subject="Please activate your account",
    body="Click to activate your account: http://example.com/activate",
    from_email="Example <admin@example.com>",
    to=["New User <user1@example.com>", "account.manager@example.com"],
    reply_to=["Helpdesk <support@example.com>"])

# Include an inline image in the html:
logo_cid = attach_inline_image_file(msg, "/path/to/logo.jpg")
html = """<img alt="Logo" src="cid:{logo_cid}">
          <p>Please <a href="http://example.com/activate">activate</a>
          your account</p>""".format(logo_cid=logo_cid)
msg.attach_alternative(html, "text/html")

# Optional Anymail extensions:
msg.metadata = {"user_id": "8675309", "experiment_variation": 1}
msg.tags = ["activation", "onboarding"]
msg.track_clicks = True

# Send it:
msg.send()
```

Problems? We have some *Troubleshooting* info that may help.

Now what?

Now that you've got Anymail working, you might be interested in:

- *Sending email with Anymail*
- *Receiving inbound email*
- *ESP-specific information*
- *All the docs*

1.2 Installation and configuration

1.2.1 Installing Anymail

To use Anymail in your Django project:

1. Install the django-anymail app. It's easiest to install from PyPI using pip:

```
$ pip install django-anymail[sendgrid,sparkpost]
```

The `[sendgrid,sparkpost]` part of that command tells pip you also want to install additional packages required for those ESPs. You can give one or more comma-separated, lowercase ESP names. (Most ESPs don't have additional requirements, so you can often just skip this. Or change your mind later. Anymail will let you know if there are any missing dependencies when you try to use it.)

2. Edit your Django project's `settings.py`, and add `anymail` to your `INSTALLED_APPS` (anywhere in the list):

```
INSTALLED_APPS = [
    # ...
    "anymail",
    # ...
]
```

3. Also in `settings.py`, add an *ANYMAIL* settings dict, substituting the appropriate settings for your ESP. E.g.:

```
ANYMAIL = {
    "MAILGUN_API_KEY": "<your Mailgun key>",
}
```

The exact settings vary by ESP. See the *supported ESPs* section for specifics.

Then continue with either or both of the next two sections, depending on which Anymail features you want to use.

1.2.2 Configuring Django's email backend

To use Anymail for *sending* email from Django, make additional changes in your project's `settings.py`. (Skip this section if you are only planning to *receive* email.)

1. Change your existing Django `EMAIL_BACKEND` to the Anymail backend for your ESP. For example, to send using Mailgun by default:

```
EMAIL_BACKEND = "anymail.backends.mailgun.EmailBackend"
```

(`EMAIL_BACKEND` sets Django's default for sending emails; you can also use *multiple Anymail backends* to send particular messages through different ESPs.)

2. If you don't already have `DEFAULT_FROM_EMAIL` and `SERVER_EMAIL` in your settings, this is a good time to add them. (Django's defaults are "webmaster@localhost" and "root@localhost", respectively, and most ESPs won't allow sending from those addresses.)

With the settings above, you are ready to send outgoing email through your ESP. If you also want to enable status tracking or inbound handling, continue with the settings below. Otherwise, skip ahead to *Sending email*.

1.2.3 Configuring tracking and inbound webhooks

Anymail can optionally connect to your ESP's event webhooks to notify your app of:

- status tracking events for sent email, like bounced or rejected messages, successful delivery, message opens and clicks, etc.
- inbound message events, if you are set up to receive email through your ESP

Skip this section if you won't be using Anymail's webhooks.

Warning: Webhooks are ordinary urls, and are wide open to the internet. You must use care to **avoid creating security vulnerabilities** that could expose your users' emails and other private information, or subject your app to malicious input data.

At a minimum, your site should **use https** and you should configure a **webhook secret** as described below.

See *Securing webhooks* for additional information.

If you want to use Anymail's inbound or tracking webhooks:

1. In your `settings.py`, add *WEBHOOK_SECRET* to the `ANYMAIL` block:

   ```
   ANYMAIL = {
       ...
       'WEBHOOK_SECRET': '<a random string>:<another random string>',
   }
   ```

   This setting should be a string with two sequences of random characters, separated by a colon. It is used as a shared secret, known only to your ESP and your Django app, to ensure nobody else can call your webhooks.

   We suggest using 16 characters (or more) for each half of the secret. Always generate a new, random secret just for this purpose. (*Don't* use your Django secret key or ESP's API key.)

   An easy way to generate a random secret is to run this command in a shell:

   ```
   $ python -c "from django.utils import crypto; print(':'.join(crypto.get_random_
   ↪string(16) for _ in range(2)))"
   ```

   (This setting is actually an HTTP basic auth string. You can also set it to a list of auth strings, to simplify credential rotation or use different auth with different ESPs. See *ANYMAIL_WEBHOOK_SECRET* in the *Securing webhooks* docs for more details.)

2. In your project's `urls.py`, add routing for the Anymail webhook urls:

   ```
   from django.conf.urls import include, url

   urlpatterns = [
       ...
       url(r'^anymail/', include('anymail.urls')),
   ]
   ```

   (You can change the "anymail" prefix in the first parameter to `url()` if you'd like the webhooks to be served at some other URL. Just match whatever you use in the webhook URL you give your ESP in the next step.)

3. Enter the webhook URL(s) into your ESP's dashboard or control panel. In most cases, the URL will be:

   `https://`*`random`*`:`*`random`*`@`*`yoursite.example.com`*`/anymail/`*`esp`*`/`*`type`*`/`

   - "https" (rather than http) is *strongly recommended*
   - *random:random* is the WEBHOOK_SECRET string you created in step 1
   - *yoursite.example.com* is your Django site
   - "anymail" is the url prefix (from step 2)
   - *esp* is the lowercase name of your ESP (e.g., "sendgrid" or "mailgun")
   - *type* is either "tracking" for Anymail's sent-mail event tracking webhooks, or "inbound" for receiving email

Some ESPs support different webhooks for different tracking events. You can usually enter the same Anymail *tracking* webhook URL for all of them (or all that you want to receive)—but be sure to use the separate *inbound* URL for inbound webhooks. And always check the specific details for your ESP under *Supported ESPs*.

Also, some ESPs try to validate the webhook URL immediately when you enter it. If so, you'll need to deploy your Django project to your live server before you can complete this step.

Some WSGI servers may need additional settings to pass HTTP authorization headers through to Django. For example, Apache with mod_wsgi requires `WSGIPassAuthorization On`, else Anymail will complain about "missing or invalid basic auth" when your webhook is called.

See *Tracking sent mail status* for information on creating signal handlers and the status tracking events you can receive. See *Receiving mail* for information on receiving inbound message events.

1.2.4 Anymail settings reference

You can add Anymail settings to your project's `settings.py` either as a single `ANYMAIL` dict, or by breaking out individual settings prefixed with `ANYMAIL_`. So this settings dict:

```
ANYMAIL = {
    "MAILGUN_API_KEY": "12345",
    "SEND_DEFAULTS": {
        "tags": ["myapp"]
    },
}
```

. . . is equivalent to these individual settings:

```
ANYMAIL_MAILGUN_API_KEY = "12345"
ANYMAIL_SEND_DEFAULTS = {"tags": ["myapp"]}
```

In addition, for some ESP settings like API keys, Anymail will look for a setting without the `ANYMAIL_` prefix if it can't find the Anymail one. (This can be helpful if you are using other Django apps that work with the same ESP.)

```
MAILGUN_API_KEY = "12345"  # used only if neither ANYMAIL["MAILGUN_API_KEY"]
                           # nor ANYMAIL_MAILGUN_API_KEY have been set
```

Finally, for complex use cases, you can override most settings on a per-instance basis by providing keyword args where the instance is initialized (e.g., in a `get_connection()` call to create an email backend instance, or in `View.as_view()` call to set up webhooks in a custom urls.py). To get the kwargs parameter for a setting, drop "ANYMAIL" and the ESP name, and lowercase the rest: e.g., you can override ANYMAIL_MAILGUN_API_KEY by passing `api_key="abc"` to `get_connection()`. See *Mixing email backends* for an example.

There are specific Anymail settings for each ESP (like API keys and urls). See the *supported ESPs* section for details. Here are the other settings Anymail supports:

IGNORE_RECIPIENT_STATUS

Set to `True` to disable `AnymailRecipientsRefused` exceptions on invalid or rejected recipients. (Default `False`.) See *Refused recipients*.

```
ANYMAIL = {
    ...
    "IGNORE_RECIPIENT_STATUS": True,
}
```

SEND_DEFAULTS and *ESP*_SEND_DEFAULTS‘

A `dict` of default options to apply to all messages sent through Anymail. See *Global send defaults*.

IGNORE_UNSUPPORTED_FEATURES

Whether Anymail should raise `AnymailUnsupportedFeature` errors for email with features that can't be accurately communicated to the ESP. Set to `True` to ignore these problems and send the email anyway. See *Unsupported features*. (Default `False`.)

WEBHOOK_SECRET

A `'random:random'` shared secret string. Anymail will reject incoming webhook calls from your ESP that don't include this authentication. You can also give a list of shared secret strings, and Anymail will allow ESP webhook calls that match any of them (to facilitate credential rotation). See *Securing webhooks*.

Default is unset, which leaves your webhooks insecure. Anymail will warn if you try to use webhooks without a shared secret.

This is actually implemented using HTTP basic authentication, and the string is technically a "username:password" format. But you should *not* use any real username or password for this shared secret.

Changed in version 1.4: The earlier WEBHOOK_AUTHORIZATION setting was renamed WEBHOOK_SECRET, so that Django error reporting sanitizes it. Support for the old name was dropped in Anymail 2.0, and if you have not yet updated your settings.py, all webhook calls will fail with a "missing or invalid basic auth" error.

REQUESTS_TIMEOUT

New in version 1.3.

For Requests-based Anymail backends, the timeout value used for all API calls to your ESP. The default is 30 seconds. You can set to a single float, a 2-tuple of floats for separate connection and read timeouts, or `None` to disable timeouts (not recommended). See Timeouts in the Requests docs for more information.

1.3 Sending email

1.3.1 Django email support

Anymail builds on Django's core email functionality. If you are already sending email using Django's default SMTP `EmailBackend`, switching to Anymail will be easy. Anymail is designed to "just work" with Django.

If you're not familiar with Django's email functions, please take a look at "`sending email`" in the Django docs first.

Anymail supports most of the functionality of Django's `EmailMessage` and `EmailMultiAlternatives` classes.

Anymail handles **all** outgoing email sent through Django's `django.core.mail` module, including `send_mail()`, `send_mass_mail()`, the `EmailMessage` class, and even `mail_admins()`. If you'd like to selectively send only some messages through Anymail, or you'd like to use different ESPs for particular messages, there are ways to use *multiple email backends*.

HTML email

To send an HTML message, you can simply use Django's `send_mail()` function with the `html_message` parameter:

```
from django.core.mail import send_mail

send_mail("Subject", "text body", "from@example.com",
          ["to@example.com"], html_message="<html>html body</html>")
```

However, many Django email capabilities – and additional Anymail features – are only available when working with an `EmailMultiAlternatives` object. Use its `attach_alternative()` method to send HTML:

```
from django.core.mail import EmailMultiAlternatives

msg = EmailMultiAlternatives("Subject", "text body",
                             "from@example.com", ["to@example.com"])
msg.attach_alternative("<html>html body</html>", "text/html")
# you can set any other options on msg here, then...
msg.send()
```

It's good practice to send equivalent content in your plain-text body and the html version.

Attachments

Anymail will send a message's attachments to your ESP. You can add attachments with the `attach()` or `attach_file()` methods of Django's `EmailMessage`.

Note that some ESPs impose limits on the size and type of attachments they will send.

Inline images

If your message has any attachments with *`Content-Disposition: inline`* headers, Anymail will tell your ESP to treat them as inline rather than ordinary attached files. If you want to reference an attachment from an `<img>` in your HTML source, the attachment also needs a *`Content-ID`* header.

Anymail comes with *`attach_inline_image()`* and *`attach_inline_image_file()`* convenience functions that do the right thing. See *Inline images* in the "Anymail additions" section.

(If you prefer to do the work yourself, Python's `MIMEImage` and `add_header()` should be helpful.)

Even if you mark an attachment as inline, some email clients may decide to also display it as an attachment. This is largely outside your control.

Changed in version 4.3: For convenience, Anymail will treat an attachment with a *`Content-ID`* but no *`Content-Disposition`* as inline. (Many—though not all—email clients make the same assumption. But to ensure consistent behavior with non-Anymail email backends, you should always set *both* *`Content-ID`* and *`Content-Disposition: inline`* headers for inline images. Or just use Anymail's *inline image helpers*, which handle this for you.)

Additional headers

Anymail passes additional headers to your ESP. (Some ESPs may limit which headers they'll allow.) EmailMessage expects a `dict` of headers:

```
# Use `headers` when creating an EmailMessage
msg = EmailMessage( ...
    headers={
        "List-Unsubscribe": unsubscribe_url,
        "X-Example-Header": "myapp",
    }
)

# Or use the `extra_headers` attribute later
msg.extra_headers["In-Reply-To"] = inbound_msg["Message-ID"]
```

Anymail treats header names as case-*insensitive* (because that's how email handles them). If you supply multiple headers that differ only in case, only one of them will make it into the resulting email.

Django's default `SMTP EmailBackend` has special handling for certain headers. Anymail replicates its behavior for compatibility:

- If you supply a "Reply-To" header, it will *override* the message's `reply_to` attribute.
- If you supply a "From" header, it will override the message's `from_email` and become the *`From`* field the recipient sees. In addition, the original `from_email` value will be used as the message's *`envelope_sender`*, which becomes the *`Return-Path`* at the recipient end. (Only if your ESP supports altering envelope sender, otherwise you'll get an *unsupported feature* error.)
- If you supply a "To" header, you'll usually get an *unsupported feature* error. With Django's SMTP EmailBackend, this can be used to show the recipient a *`To`* address that's different from the actual envelope recipients in the message's `to` list. Spoofing the *`To`* header like this is popular with spammers, and almost none of Anymail's supported ESPs allow it.

Changed in version 2.0: Improved header-handling compatibility with Django's SMTP EmailBackend.

Unsupported features

Some email capabilities aren't supported by all ESPs. When you try to send a message using features Anymail can't communicate to the current ESP, you'll get an *`AnymailUnsupportedFeature`* error, and the message won't be sent.

For example, very few ESPs support alternative message parts added with `attach_alternative()` (other than a single *`text/html`* part that becomes the HTML body). If you try to send a message with other alternative parts, Anymail will raise *`AnymailUnsupportedFeature`*. If you'd like to silently ignore *`AnymailUnsupportedFeature`* errors and send the messages anyway, set `ANYMAIL_IGNORE_UNSUPPORTED_FEATURES` to `True` in your settings.py:

```
ANYMAIL = {
    ...
    "IGNORE_UNSUPPORTED_FEATURES": True,
}
```

Refused recipients

If *all* recipients (to, cc, bcc) of a message are invalid or rejected by your ESP *at send time,* the send call will raise an *`AnymailRecipientsRefused`* error.

You can examine the message's *`anymail_status`* attribute to determine the cause of the error. (See *ESP send status*.)

If a single message is sent to multiple recipients, and *any* recipient is valid (or the message is queued by your ESP because of rate limiting or *send_at*), then this exception will not be raised. You can still examine the message's *anymail_status* property after the send to determine the status of each recipient.

You can disable this exception by setting *ANYMAIL_IGNORE_RECIPIENT_STATUS* to `True` in your settings.py, which will cause Anymail to treat any non-API-error response from your ESP as a successful send.

Note: Many ESPs don't check recipient status during the send API call. For example, Mailgun always queues sent messages, so you'll never catch `AnymailRecipientsRefused` with the Mailgun backend.

For those ESPs, use Anymail's *delivery event tracking* if you need to be notified of sends to blacklisted or invalid emails.

1.3.2 Anymail additions

Anymail normalizes several common ESP features, like adding metadata or tags to a message. It also normalizes the response from the ESP's send API.

There are three ways you can use Anymail's ESP features with your Django email:

- Just use Anymail's added attributes directly on *any* Django `EmailMessage` object (or any subclass).
- Create your email message using the *AnymailMessage* class, which exposes extra attributes for the ESP features.
- Use the *AnymailMessageMixin* to add the Anymail extras to some other EmailMessage-derived class (your own or from another Django package).

The first approach is usually the simplest. The other two can be helpful if you are working with Python development tools that offer type checking or other static code analysis.

ESP send options (AnymailMessage)

class anymail.message.**AnymailMessage**

A subclass of Django's `EmailMultiAlternatives` that exposes additional ESP functionality.

The constructor accepts any of the attributes below, or you can set them directly on the message at any time before sending:

```
from anymail.message import AnymailMessage

message = AnymailMessage(
    subject="Welcome",
    body="Welcome to our site",
    to=["New User <user1@example.com>"],
    tags=["Onboarding"],  # Anymail extra in constructor
)
# Anymail extra attributes:
message.metadata = {"onboarding_experiment": "variation 1"}
message.track_clicks = True

message.send()
status = message.anymail_status  # available after sending
status.message_id  # e.g., '<12345.67890@example.com>'
status.recipients["user1@example.com"].status  # e.g., 'queued'
```

Attributes you can add to messages

Note: Anymail looks for these attributes on **any** `EmailMessage` you send. (You don't have to use *`AnymailMessage`*.)

`envelope_sender`
New in version 2.0.

Set this to a `str` email address that should be used as the message's envelope sender. If supported by your ESP, this will become the Return-Path in the recipient's mailbox.

(Envelope sender is also known as bounce address, MAIL FROM, envelope from, unixfrom, SMTP FROM command, return path, and several other terms. Confused? Here's some good info on how envelope sender relates to return path.)

ESP support for envelope sender varies widely. Be sure to check Anymail's docs for your *specific ESP* before attempting to use it. And note that those ESPs who do support it will often use only the domain portion of the envelope sender address, overriding the part before the @ with their own encoded bounce mailbox.

[The `envelope_sender` attribute is unique to Anymail. If you also use Django's SMTP EmailBackend, you can portably control envelope sender by *instead* setting `message.extra_headers["From"]` to the desired email *header* *`From`*, and `message.from_email` to the *envelope sender*. Anymail also allows this approach, for compatibility with the SMTP EmailBackend. See the notes in Django's bug tracker.]

`metadata`
Set this to a `dict` of metadata values the ESP should store with the message, for later search and retrieval.

```
message.metadata = {"customer": customer.id,
                    "order": order.reference_number}
```

ESPs have differing restrictions on metadata content. For portability, it's best to stick to alphanumeric keys, and values that are numbers or strings.

You should format any non-string data into a string before setting it as metadata. See *Formatting merge data*.

`merge_metadata`
Set this to a `dict` of *per-recipient* metadata values the ESP should store with the message, for later search and retrieval. Each key in the dict is a recipient email (address portion only), and its value is a dict of metadata for that recipient:

```
message.to = ["wile@example.com", "Mr. Runner <rr@example.com>"]
message.merge_metadata = {
    "wile@example.com": {"customer": 123, "order": "acme-zxyw"},
    "rr@example.com": {"customer": 45678, "order": "acme-wblt"},
}
```

When `merge_metadata` is set, Anymail will use the ESP's *batch sending* option, so that each `to` recipient gets an individual message (and doesn't see the other emails on the `to` list).

All of the notes on *`metadata`* keys and value formatting also apply to `merge_metadata`. If there are conflicting keys, the `merge_metadata` values will take precedence over `metadata` for that recipient.

`tags`
Set this to a `list` of `str` tags to apply to the message (usually for segmenting ESP reporting).

```
message.tags = ["Order Confirmation", "Test Variant A"]
```

ESPs have differing restrictions on tags. For portability, it's best to stick with strings that start with an alphanumeric character. (Also, Postmark only allows a single tag per message.)

Caution: Some ESPs put *metadata* (and a recipient's *merge_metadata*) and *tags* in email headers, which are included with the email when it is delivered. Anything you put in them **could be exposed to the recipients,** so don't include sensitive data.

track_opens
Set this to `True` or `False` to override your ESP account default setting for tracking when users open a message.

```
message.track_opens = True
```

track_clicks
Set this to `True` or `False` to override your ESP account default setting for tracking when users click on a link in a message.

```
message.track_clicks = False
```

send_at
Set this to a `datetime`, `date` to have the ESP wait until the specified time to send the message. (You can also use a `float` or `int`, which will be treated as a POSIX timestamp as in `time.time()`.)

```
from datetime import datetime, timedelta
from django.utils.timezone import utc

message.send_at = datetime.now(utc) + timedelta(hours=1)
```

To avoid confusion, it's best to provide either an *aware* `datetime` (one that has its tzinfo set), or an `int` or `float` seconds-since-the-epoch timestamp.

If you set `send_at` to a `date` or a *naive* `datetime` (without a timezone), Anymail will interpret it in Django's current timezone. (Careful: `datetime.now()` returns a *naive* datetime, unless you call it with a timezone like in the example above.)

The sent message will be held for delivery by your ESP – not locally by Anymail.

esp_extra
Set this to a `dict` of additional, ESP-specific settings for the message.

Using this attribute is inherently non-portable between ESPs, and is intended as an "escape hatch" for accessing functionality that Anymail doesn't (or doesn't yet) support.

See the notes for each *specific ESP* for information on its `esp_extra` handling.

Status response from the ESP

anymail_status
Normalized response from the ESP API's send call. Anymail adds this to each `EmailMessage` as it is sent.

The value is an *AnymailStatus*. See *ESP send status* for details.

Convenience methods

(These methods are only available on *AnymailMessage* or *AnymailMessageMixin* objects. Unlike the attributes above, they can't be used on an arbitrary EmailMessage.)

attach_inline_image_file(*path*, *subtype=None*, *idstring="img"*, *domain=None*)
Attach an inline (embedded) image to the message and return its *Content-ID*.

This calls *attach_inline_image_file()* on the message. See *Inline images* for details and an example.

attach_inline_image(*content*, *filename=None*, *subtype=None*, *idstring="img"*, *domain=None*)
Attach an inline (embedded) image to the message and return its *Content-ID*.

This calls *attach_inline_image()* on the message. See *Inline images* for details and an example.

ESP send status

class anymail.message.**AnymailStatus**
When you send a message through an Anymail backend, Anymail adds an *anymail_status* attribute to the EmailMessage, with a normalized version of the ESP's response.

Anymail backends create this attribute *as they process each message.* Before that, anymail_status won't be present on an ordinary Django EmailMessage or EmailMultiAlternatives—you'll get an AttributeError if you try to access it.

This might cause problems in your test cases, because Django substitutes its own locmem EmailBackend during testing (so anymail_status never gets attached to the EmailMessage). If you run into this, you can: change your code to guard against a missing anymail_status attribute; switch from using EmailMessage to *AnymailMessage* (or the *AnymailMessageMixin*) to ensure the anymail_status attribute is always there; or substitute *Anymail's test backend* in any affected test cases.

After sending through an Anymail backend, *anymail_status* will be an object with these attributes:

message_id
The message id assigned by the ESP, or None if the send call failed.

The exact format varies by ESP. Some use a UUID or similar; some use an **RFC 2822** *Message-ID* as the id:

```
message.anymail_status.message_id
# '<20160306015544.116301.25145@example.org>'
```

Some ESPs assign a unique message ID for *each recipient* (to, cc, bcc) of a single message. In that case, message_id will be a set of all the message IDs across all recipients:

```
message.anymail_status.message_id
# set(['16fd2706-8baf-433b-82eb-8c7fada847da',
#      '886313e1-3b8a-5372-9b90-0c9aee199e5d'])
```

status
A set of send statuses, across all recipients (to, cc, bcc) of the message, or None if the send call failed.

```
message1.anymail_status.status
# set(['queued'])  # all recipients were queued
message2.anymail_status.status
# set(['rejected', 'sent'])  # at least one recipient was sent,
                             # and at least one rejected
```

(continues on next page)

(continued from previous page)

```
# This is an easy way to check there weren't any problems:
if message3.anymail_status.status.issubset({'queued', 'sent'}):
    print("ok!")
```

Anymail normalizes ESP sent status to one of these values:

- `'sent'` the ESP has sent the message (though it may or may not end up delivered)
- `'queued'` the ESP has accepted the message and will try to send it asynchronously
- `'invalid'` the ESP considers the sender or recipient email invalid
- `'rejected'` the recipient is on an ESP blacklist (unsubscribe, previous bounces, etc.)
- `'failed'` the attempt to send failed for some other reason
- `'unknown'` anything else

Not all ESPs check recipient emails during the send API call – some simply queue the message, and report problems later. In that case, you can use Anymail's *Tracking sent mail status* features to be notified of delivery status events.

`recipients`
A `dict` of per-recipient message ID and status values.

The dict is keyed by each recipient's base email address (ignoring any display name). Each value in the dict is an object with *`status`* and *`message_id`* properties:

```
message = EmailMultiAlternatives(
    to=["you@example.com", "Me <me@example.com>"],
    subject="Re: The apocalypse")
message.send()

message.anymail_status.recipients["you@example.com"].status
# 'sent'
message.anymail_status.recipients["me@example.com"].status
# 'queued'
message.anymail_status.recipients["me@example.com"].message_id
# '886313e1-3b8a-5372-9b90-0c9aee199e5d'
```

Will be an empty dict if the send call failed.

`esp_response`
The raw response from the ESP API call. The exact type varies by backend. Accessing this is inherently non-portable.

```
# This will work with a requests-based backend:
message.anymail_status.esp_response.json()
```

Inline images

Anymail includes convenience functions to simplify attaching inline images to email.

These functions work with *any* Django `EmailMessage` – they're not specific to Anymail email backends. You can use them with messages sent through Django's SMTP backend or any other that properly supports MIME attachments.

(Both functions are also available as convenience methods on Anymail's *`AnymailMessage`* and *`AnymailMessageMixin`* classes.)

anymail.message.**attach_inline_image_file**(*message*, *path*, *subtype=None*, *idstring="img"*, *domain=None*)

Attach an inline (embedded) image to the message and return its *Content-ID*.

In your HTML message body, prefix the returned id with `cid:` to make an `<img>` src attribute:

```
from django.core.mail import EmailMultiAlternatives
from anymail.message import attach_inline_image_file

message = EmailMultiAlternatives( ... )
cid = attach_inline_image_file(message, 'path/to/picture.jpg')
html = '... <img alt="Picture" src="cid:%s"> ...' % cid
message.attach_alternative(html, 'text/html')

message.send()
```

`message` must be an `EmailMessage` (or subclass) object.

`path` must be the pathname to an image file. (Its basename will also be used as the attachment's filename, which may be visible in some email clients.)

`subtype` is an optional MIME *image* subtype, e.g., `"png"` or `"jpg"`. By default, this is determined automatically from the content.

`idstring` and `domain` are optional, and are passed to Python's `make_msgid()` to generate the *Content-ID*. Generally the defaults should be fine.

Changed in version 4.0: If you don't supply a `domain`, Anymail will use the simple string "inline" rather than `make_msgid()`'s default local hostname. This avoids a problem with ESPs that confuse *Content-ID* and attachment filename: if your local server's hostname ends in ".com", Gmail could block messages with inline attachments generated by earlier Anymail versions and sent through these ESPs.

anymail.message.**attach_inline_image**(*message*, *content*, *filename=None*, *subtype=None*, *idstring="img"*, *domain=None*)

This is a version of *attach_inline_image_file()* that accepts raw image data, rather than reading it from a file.

`message` must be an `EmailMessage` (or subclass) object.

`content` must be the binary image data

`filename` is an optional `str` that will be used as as the attachment's filename – e.g., `"picture.jpg"`. This may be visible in email clients that choose to display the image as an attachment as well as making it available for inline use (this is up to the email client). It should be a base filename, without any path info.

`subtype`, `idstring` and `domain` are as described in *attach_inline_image_file()*

Global send defaults

In your `settings.py`, you can set `ANYMAIL_SEND_DEFAULTS` to a `dict` of default options that will apply to all messages sent through Anymail:

```
ANYMAIL = {
    ...
    "SEND_DEFAULTS": {
        "metadata": {"district": "North", "source": "unknown"},
        "tags": ["myapp", "version3"],
        "track_clicks": True,
        "track_opens": True,
```

(continues on next page)

(continued from previous page)

```
    },
}
```

At send time, the attributes on each `EmailMessage` get merged with the global send defaults. For example, with the settings above:

```
message = AnymailMessage(...)
message.tags = ["welcome"]
message.metadata = {"source": "Ads", "user_id": 12345}
message.track_clicks = False

message.send()
# will send with:
#   tags: ["myapp", "version3", "welcome"] (merged with defaults)
#   metadata: {"district": "North", "source": "Ads", "user_id": 12345}␣
↪(merged)
#   track_clicks: False (message overrides defaults)
#   track_opens: True (from the defaults)
```

To prevent a message from using a particular global default, set that attribute to `None`. (E.g., `message.tags = None` will send the message with no tags, ignoring the global default.)

Anymail's send defaults actually work for all `django.core.mail.EmailMessage` attributes. So you could set `"bcc": ["always-copy@example.com"]` to add a bcc to every message. (You could even attach a file to every message – though your recipients would probably find that annoying!)

You can also set ESP-specific global defaults. If there are conflicts, the ESP-specific value will override the main `SEND_DEFAULTS`:

```
ANYMAIL = {
    ...
    "SEND_DEFAULTS": {
        "tags": ["myapp", "version3"],
    },
    "POSTMARK_SEND_DEFAULTS": {
        # Postmark only supports a single tag
        "tags": ["version3"],  # overrides SEND_DEFAULTS['tags'] (not merged!
↪)
    },
    "MAILGUN_SEND_DEFAULTS": {
        "esp_extra": {"o:dkim": "no"},  # Disable Mailgun DKIM signatures
    },
}
```

AnymailMessageMixin

class anymail.message.**AnymailMessageMixin**

Mixin class that adds Anymail's ESP extra attributes and convenience methods to other `EmailMessage` subclasses.

For example, with the django-mail-templated package's custom EmailMessage:

```
from anymail.message import AnymailMessageMixin
from mail_templated import EmailMessage
```

(continues on next page)

(continued from previous page)

```
class TemplatedAnymailMessage(AnymailMessageMixin, EmailMessage):
    """
    An EmailMessage that supports both Mail-Templated
    and Anymail features
    """
    pass

msg = TemplatedAnymailMessage(
    template_name="order_confirmation.tpl",  # Mail-Templated arg
    track_opens=True,  # Anymail arg
    ...
)
msg.context = {"order_num": "12345"}  # Mail-Templated attribute
msg.tags = ["templated"]  # Anymail attribute
```

1.3.3 Batch sending/merge and ESP templates

If your ESP offers templates and batch-sending/merge capabilities, Anymail can simplify using them in a portable way. Anymail doesn't translate template syntax between ESPs, but it does normalize using templates and providing merge data for batch sends.

Here's an example using both an ESP stored template and merge data:

```
from django.core.mail import EmailMessage

message = EmailMessage(
    subject=None,  # use the subject in our stored template
    from_email="marketing@example.com",
    to=["Wile E. <wile@example.com>", "rr@example.com"])
message.template_id = "after_sale_followup_offer"  # use this ESP stored template
message.merge_data = {  # per-recipient data to merge into the template
    'wile@example.com': {'NAME': "Wile E.",
                         'OFFER': "15% off anvils"},
    'rr@example.com':   {'NAME': "Mr. Runner"},
}
message.merge_global_data = {  # merge data for all recipients
    'PARTNER': "Acme, Inc.",
    'OFFER': "5% off any Acme product",  # a default if OFFER missing for recipient
}
message.send()
```

The message's *template_id* identifies a template stored at your ESP which provides the message body and subject. (Assuming the ESP supports those features.)

The message's *merge_data* supplies the per-recipient data to substitute for merge fields in your template. Setting this attribute also lets Anymail know it should use the ESP's *batch sending* feature to deliver separate, individually-customized messages to each address on the "to" list. (Again, assuming your ESP supports that.)

Note: Templates and batch sending capabilities can vary widely between ESPs, as can the syntax for merge fields. Be sure to read the notes for *your specific ESP*, and test carefully with a small recipient list before launching a gigantic batch send.

Although related and often used together, *ESP stored templates* and *merge data* are actually independent features. For example, some ESPs will let you use merge field syntax directly in your `EmailMessage` body, so you can do

customized batch sending without needing to define a stored template at the ESP.

ESP stored templates

Many ESPs support transactional email templates that are stored and managed within your ESP account. To use an ESP stored template with Anymail, set *template_id* on an EmailMessage.

AnymailMessage.**template_id**
: The identifier of the ESP stored template you want to use. For most ESPs, this is a str name or unique id. (See the notes for your *specific ESP*.)

```
message.template_id = "after_sale_followup_offer"
```

With most ESPs, using a stored template will ignore any body (plain-text or HTML) from the EmailMessage object.

A few ESPs also allow you to define the message's subject as part of the template, but any subject you set on the EmailMessage will override the template subject. To use the subject stored with the ESP template, set the message's subject to None:

```
message.subject = None  # use subject from template (if supported)
```

Similarly, some ESPs can also specify the "from" address in the template definition. Set message.from_email = None to use the template's "from." (You must set this attribute *after* constructing an EmailMessage object; passing from_email=None to the constructor will use Django's DEFAULT_FROM_EMAIL setting, overriding your template value.)

Batch sending with merge data

Several ESPs support "batch transactional sending," where a single API call can send messages to multiple recipients. The message is customized for each email on the "to" list by merging per-recipient data into the body and other message fields.

To use batch sending with Anymail (for ESPs that support it):

- Use "merge fields" (sometimes called "substitution variables" or similar) in your message. This could be in an *ESP stored template* referenced by *template_id*, or with some ESPs you can use merge fields directly in your EmailMessage (meaning the message itself is treated as an on-the-fly template).
- Set the message's *merge_data* attribute to define merge field substitutions for each recipient, and optionally set *merge_global_data* to defaults or values to use for all recipients.
- Specify all of the recipients for the batch in the message's to list.

Caution: It's critical to set the *merge_data* (or *merge_metadata*) attribute: this is how Anymail recognizes the message as a batch send.

When you provide merge_data, Anymail will tell the ESP to send an individual customized message to each "to" address. Without it, you may get a single message to everyone, exposing all of the email addresses to all recipients. (If you don't have any per-recipient customizations, but still want individual messages, just set merge_data to an empty dict.)

The exact syntax for merge fields varies by ESP. It might be something like *|NAME|* or -name- or <%name%>. (Check the notes for *your ESP*, and remember you'll need to change the template if you later switch ESPs.)

AnymailMessage.**merge_data**
: A `dict` of *per-recipient* template substitution/merge data. Each key in the dict is a recipient email address, and its value is a `dict` of merge field names and values to use for that recipient:

```
message.merge_data = {
    'wile@example.com': {'NAME': "Wile E.",
                         'OFFER': "15% off anvils"},
    'rr@example.com':   {'NAME': "Mr. Runner",
                         'OFFER': "instant tunnel paint"},
}
```

When *merge_data* is set, Anymail will use the ESP's batch sending option, so that each `to` recipient gets an individual message (and doesn't see the other emails on the `to` list).

AnymailMessage.**merge_global_data**
: A `dict` of template substitution/merge data to use for *all* recipients. Keys are merge field names in your message template:

```
message.merge_global_data = {
    'PARTNER': "Acme, Inc.",
    'OFFER': "5% off any Acme product",  # a default OFFER
}
```

Merge data values must be strings. (Some ESPs also allow other JSON-serializable types like lists or dicts.) See *Formatting merge data* for more information.

Like all *Anymail additions*, you can use these extended template and merge attributes with any `EmailMessage` or subclass object. (It doesn't have to be an *AnymailMessage*.)

Tip: you can add `merge_global_data` to your global Anymail *send defaults* to supply merge data available to all batch sends (e.g, site name, contact info). The global defaults will be merged with any per-message `merge_global_data`.

Formatting merge data

If you're using a `date`, `datetime`, `Decimal`, or anything other than strings and integers, you'll need to format them into strings for use as merge data:

```
product = Product.objects.get(123)  # A Django model
total_cost = Decimal('19.99')
ship_date = date(2015, 11, 18)

# Won't work -- you'll get "not JSON serializable" errors at send time:
message.merge_global_data = {
    'PRODUCT': product,
    'TOTAL_COST': total_cost,
    'SHIP_DATE': ship_date
}

# Do something this instead:
message.merge_global_data = {
    'PRODUCT': product.name,  # assuming name is a CharField
    'TOTAL_COST': "%.2f" % total_cost,
    'SHIP_DATE': ship_date.strftime('%B %d, %Y')  # US-style "March 15, 2015"
}
```

These are just examples. You'll need to determine the best way to format your merge data as strings.

Although floats are usually allowed in merge data, you'll generally want to format them into strings yourself to avoid surprises with floating-point precision.

Anymail will raise *AnymailSerializationError* if you attempt to send a message with merge data (or metadata) that can't be sent to your ESP.

ESP templates vs. Django templates

ESP templating languages are generally proprietary, which makes them inherently non-portable.

Anymail only exposes the stored template capabilities that your ESP already offers, and then simplifies providing merge data in a portable way. It won't translate between different ESP template syntaxes, and it can't do a batch send if your ESP doesn't support it.

There are two common cases where ESP template and merge features are particularly useful with Anymail:

- When the people who develop and maintain your transactional email templates are different from the people who maintain your Django page templates. (For example, you use a single ESP for both marketing and transactional email, and your marketing team manages all the ESP email templates.)
- When you want to use your ESP's batch-sending capabilities for performance reasons, where a single API call can trigger individualized messages to hundreds or thousands of recipients. (For example, sending a daily batch of shipping notifications.)

If neither of these cases apply, you may find that *using Django templates* can be a more portable and maintainable approach for building transactional email.

1.3.4 Tracking sent mail status

Anymail provides normalized handling for your ESP's event-tracking webhooks. You can use this to be notified when sent messages have been delivered, bounced, been opened or had links clicked, among other things.

Webhook support is optional. If you haven't yet, you'll need to *configure webhooks* in your Django project. (You may also want to review *Securing webhooks*.)

Once you've enabled webhooks, Anymail will send a `anymail.signals.tracking` custom Django `signal` for each ESP tracking event it receives. You can connect your own receiver function to this signal for further processing.

Be sure to read Django's listening to signals docs for information on defining and connecting signal receivers.

Example:

```
from anymail.signals import tracking
from django.dispatch import receiver

@receiver(tracking)  # add weak=False if inside some other function/class
def handle_bounce(sender, event, esp_name, **kwargs):
    if event.event_type == 'bounced':
        print("Message %s to %s bounced" % (
              event.message_id, event.recipient))

@receiver(tracking)
def handle_click(sender, event, esp_name, **kwargs):
    if event.event_type == 'clicked':
        print("Recipient %s clicked url %s" % (
              event.recipient, event.click_url))
```

You can define individual signal receivers, or create one big one for all event types, which ever you prefer. You can even handle the same event in multiple receivers, if that makes your code cleaner. These *signal receiver functions* are documented in more detail below.

Note that your tracking signal recevier(s) will be called for all tracking webhook types you've enabled at your ESP, so you should always check the `event_type` as shown in the examples above to ensure you're processing the expected events.

Some ESPs batch up multiple events into a single webhook call. Anymail will invoke your signal receiver once, separately, for each event in the batch.

Normalized tracking event

class anymail.signals.**AnymailTrackingEvent**

The `event` parameter to Anymail's `tracking` *signal receiver* is an object with the following attributes:

event_type

A normalized `str` identifying the type of tracking event.

Note: Most ESPs will send some, but *not all* of these event types. Check the *specific ESP* docs for more details. In particular, very few ESPs implement the "sent" and "delivered" events.

One of:

- `'queued'`: the ESP has accepted the message and will try to send it (possibly at a later time).
- `'sent'`: the ESP has sent the message (though it may or may not get successfully delivered).
- `'rejected'`: the ESP refused to send the messsage (e.g., because of a suppression list, ESP policy, or invalid email). Additional info may be in `reject_reason`.
- `'failed'`: the ESP was unable to send the message (e.g., because of an error rendering an ESP template)
- `'bounced'`: the message was rejected or blocked by receiving MTA (message transfer agent—the receiving mail server).
- `'deferred'`: the message was delayed by in transit (e.g., because of a transient DNS problem, a full mailbox, or certain spam-detection strategies). The ESP will keep trying to deliver the message, and should generate a separate `'bounced'` event if later it gives up.
- `'delivered'`: the message was accepted by the receiving MTA. (This does not guarantee the user will see it. For example, it might still be classified as spam.)
- `'autoresponded'`: a robot sent an automatic reply, such as a vacation notice, or a request to prove you're a human.
- `'opened'`: the user opened the message (used with your ESP's `track_opens` feature).
- `'clicked'`: the user clicked a link in the message (used with your ESP's `track_clicks` feature).
- `'complained'`: the recipient reported the message as spam.
- `'unsubscribed'`: the recipient attempted to unsubscribe (when you are using your ESP's subscription management features).
- `'subscribed'`: the recipient attempted to subscribe to a list, or undo an earlier unsubscribe (when you are using your ESP's subscription management features).
- `'unknown'`: anything else. Anymail isn't able to normalize this event, and you'll need to examine the raw `esp_event` data.

message_id
A `str` unique identifier for the message, matching the *`message.anymail_status.message_id`* attribute from when the message was sent.

The exact format of the string varies by ESP. (It may or may not be an actual "Message-ID", and is often some sort of UUID.)

timestamp
A `datetime` indicating when the event was generated. (The timezone is often UTC, but the exact behavior depends on your ESP and account settings. Anymail ensures that this value is an *aware* datetime with an accurate timezone.)

event_id
A `str` unique identifier for the event, if available; otherwise `None`. Can be used to avoid processing the same event twice. Exact format varies by ESP, and not all ESPs provide an event_id for all event types.

recipient
The `str` email address of the recipient. (Just the "recipient@example.com" portion.)

metadata
A `dict` of unique data attached to the message. Will be empty if the ESP doesn't provide metadata with its tracking events. (See *`AnymailMessage.metadata`*.)

tags
A `list` of `str` tags attached to the message. Will be empty if the ESP doesn't provide tags with its tracking events. (See *`AnymailMessage.tags`*.)

reject_reason
For `'bounced'` and `'rejected'` events, a normalized `str` giving the reason for the bounce/rejection. Otherwise `None`. One of:

- `'invalid'`: bad email address format.
- `'bounced'`: bounced recipient. (In a `'rejected'` event, indicates the recipient is on your ESP's prior-bounces suppression list.)
- `'timed_out'`: your ESP is giving up after repeated transient delivery failures (which may have shown up as `'deferred'` events).
- `'blocked'`: your ESP's policy prohibits this recipient.
- `'spam'`: the receiving MTA or recipient determined the message is spam. (In a `'rejected'` event, indicates the recipient is on your ESP's prior-spam-complaints suppression list.)
- `'unsubscribed'`: the recipient is in your ESP's unsubscribed suppression list.
- `'other'`: some other reject reason; examine the raw *`esp_event`*.
- `None`: Anymail isn't able to normalize a reject/bounce reason for this ESP.

Note: Not all ESPs provide all reject reasons, and this area is often under-documented by the ESP. Anymail does its best to interpret the ESP event, but you may find (e.g.,) that it will report `'timed_out'` for one ESP, and `'bounced'` for another, sending to the same non-existent mailbox.

We appreciate *bug reports* with the raw *`esp_event`* data in cases where Anymail is getting it wrong.

---

description
If available, a `str` with a (usually) human-readable description of the event. Otherwise `None`. For example, might explain why an email has bounced. Exact format varies by ESP (and sometimes event type).

mta_response
: If available, a `str` with a raw (intended for email administrators) response from the receiving MTA. Otherwise `None`. Often includes SMTP response codes, but the exact format varies by ESP (and sometimes receiving MTA).

user_agent
: For `'opened'` and `'clicked'` events, a `str` identifying the browser and/or email client the user is using, if available. Otherwise `None`.

click_url
: For `'clicked'` events, the `str` url the user clicked. Otherwise `None`.

esp_event
: The "raw" event data from the ESP, deserialized into a python data structure. For most ESPs this is either parsed JSON (as a `dict`), or HTTP POST fields (as a Django `QueryDict`).

  This gives you (non-portable) access to additional information provided by your ESP. For example, some ESPs include geo-IP location information with open and click events.

Signal receiver functions

Your Anymail signal receiver must be a function with this signature:

def my_handler(sender, event, esp_name, **kwargs):
: (You can name it anything you want.)

  Parameters

  - **sender** (*class*) – The source of the event. (One of the `anymail.webhook.*` View classes, but you generally won't examine this parameter; it's required by Django's signal mechanism.)
  - **event** (`AnymailTrackingEvent`) – The normalized tracking event. Almost anything you'd be interested in will be in here.
  - **esp_name** (*str*) – e.g., "SendMail" or "Postmark". If you are working with multiple ESPs, you can use this to distinguish ESP-specific handling in your shared event processing.
  - ****kwargs** – Required by Django's signal mechanism (to support future extensions).

  Returns nothing

  Raises any exceptions in your signal receiver will result in a 400 HTTP error to the webhook. See discussion below.

If (any of) your signal receivers raise an exception, Anymail will discontinue processing the current batch of events and return an HTTP 400 error to the ESP. Most ESPs respond to this by re-sending the event(s) later, a limited number of times.

This is the desired behavior for transient problems (e.g., your Django database being unavailable), but can cause confusion in other error cases. You may want to catch some (or all) exceptions in your signal receiver, log the problem for later follow up, and allow Anymail to return the normal 200 success response to your ESP.

Some ESPs impose strict time limits on webhooks, and will consider them failed if they don't respond within (say) five seconds. And will retry sending the "failed" events, which could cause duplicate processing in your code. If your signal receiver code might be slow, you should instead queue the event for later, asynchronous processing (e.g., using something like Celery).

If your signal receiver function is defined within some other function or instance method, you *must* use the `weak=False` option when connecting it. Otherwise, it might seem to work at first, but will unpredictably stop

being called at some point—typically on your production server, in a hard-to-debug way. See Django's listening to signals docs for more information.

1.3.5 Pre- and post-send signals

Anymail provides *pre-send* and *post-send* signals you can connect to trigger actions whenever messages are sent through an Anymail backend.

Be sure to read Django's listening to signals docs for information on defining and connecting signal receivers.

Pre-send signal

You can use Anymail's *pre_send* signal to examine or modify messages before they are sent. For example, you could implement your own email suppression list:

```
from anymail.exceptions import AnymailCancelSend
from anymail.signals import pre_send
from django.dispatch import receiver
from email.utils import parseaddr

from your_app.models import EmailBlockList

@receiver(pre_send)
def filter_blocked_recipients(sender, message, **kwargs):
    # Cancel the entire send if the from_email is blocked:
    if not ok_to_send(message.from_email):
        raise AnymailCancelSend("Blocked from_email")
    # Otherwise filter the recipients before sending:
    message.to = [addr for addr in message.to if ok_to_send(addr)]
    message.cc = [addr for addr in message.cc if ok_to_send(addr)]

def ok_to_send(addr):
    # This assumes you've implemented an EmailBlockList model
    # that holds emails you want to reject...
    name, email = parseaddr(addr)  # just want the <email> part
    try:
        EmailBlockList.objects.get(email=email)
        return False  # in the blocklist, so *not* OK to send
    except EmailBlockList.DoesNotExist:
        return True  # *not* in the blocklist, so OK to send
```

Any changes you make to the message in your pre-send signal receiver will be reflected in the ESP send API call, as shown for the filtered "to" and "cc" lists above. Note that this will modify the original EmailMessage (not a copy)—be sure this won't confuse your sending code that created the message.

If you want to cancel the message altogether, your pre-send receiver function can raise an `AnymailCancelSend` exception, as shown for the "from_email" above. This will silently cancel the send without raising any other errors.

`anymail.signals.`**`pre_send`**

Signal delivered before each EmailMessage is sent.

Your pre_send receiver must be a function with this signature:

`def my_pre_send_handler(sender, message, esp_name, **kwargs):`

(You can name it anything you want.)

Parameters

- **sender** (*class*) – The Anymail backend class processing the message. This parameter is required by Django's signal mechanism, and despite the name has nothing to do with the *email message's* sender. (You generally won't need to examine this parameter.)
- **message** (*EmailMessage*) – The message being sent. If your receiver modifies the message, those changes will be reflected in the ESP send call.
- **esp_name** (*str*) – The name of the ESP backend in use (e.g., "SendGrid" or "Mailgun").
- ****kwargs** – Required by Django's signal mechanism (to support future extensions).

Raises `anymail.exceptions.AnymailCancelSend` if your receiver wants to cancel this message without causing errors or interrupting a batch send.

Post-send signal

You can use Anymail's *post_send* signal to examine messages after they are sent. This is useful to centralize handling of the *sent status* for all messages.

For example, you could implement your own ESP logging dashboard (perhaps combined with Anymail's *event-tracking webhooks*):

```
from anymail.signals import post_send
from django.dispatch import receiver

from your_app.models import SentMessage

@receiver(post_send)
def log_sent_message(sender, message, status, esp_name, **kwargs):
    # This assumes you've implemented a SentMessage model for tracking sends.
    # status.recipients is a dict of email: status for each recipient
    for email, recipient_status in status.recipients.items():
        SentMessage.objects.create(
            esp=esp_name,
            message_id=recipient_status.message_id,  # might be None if send failed
            email=email,
            subject=message.subject,
            status=recipient_status.status,  # 'sent' or 'rejected' or ...
        )
```

`anymail.signals.`**post_send**

Signal delivered after each EmailMessage is sent.

If you register multiple post-send receivers, Anymail will ensure that all of them are called, even if one raises an error.

Your post_send receiver must be a function with this signature:

def my_post_send_handler(sender, message, status, esp_name, **kwargs):

(You can name it anything you want.)

Parameters

- **sender** (*class*) – The Anymail backend class processing the message. This parameter is required by Django's signal mechanism, and despite the name has nothing to do with the *email message's* sender. (You generally won't need to examine this parameter.)
- **message** (*EmailMessage*) – The message that was sent. You should not modify this in a post-send receiver.

- **status** (AnymailStatus) – The normalized response from the ESP send call. (Also available as *message.anymail_status*.)
- **esp_name** (*str*) – The name of the ESP backend in use (e.g., "SendGrid" or "Mailgun").
- ****kwargs** – Required by Django's signal mechanism (to support future extensions).

1.3.6 Exceptions

exception anymail.exceptions.**AnymailUnsupportedFeature**

If the email tries to use features that aren't supported by the ESP, the send call will raise an AnymailUnsupportedFeature error, and the message won't be sent. See *Unsupported features*.

You can disable this exception (ignoring the unsupported features and sending the message anyway, without them) by setting *ANYMAIL_IGNORE_UNSUPPORTED_FEATURES* to True.

exception anymail.exceptions.**AnymailRecipientsRefused**

Raised when *all* recipients (to, cc, bcc) of a message are invalid or rejected by your ESP *at send time.* See *Refused recipients*.

You can disable this exception by setting *ANYMAIL_IGNORE_RECIPIENT_STATUS* to True in your settings.py, which will cause Anymail to treat any non-*AnymailAPIError* response from your ESP as a successful send.

exception anymail.exceptions.**AnymailAPIError**

If the ESP's API fails or returns an error response, the send call will raise an AnymailAPIError.

The exception's status_code and response attributes may help explain what went wrong. (Tip: you may also be able to check the API log in your ESP's dashboard. See *Troubleshooting*.)

In production, it's not unusual for sends to occasionally fail due to transient connectivity problems, ESP maintenance, or other operational issues. Typically these failures have a 5xx status_code. See *Handling transient errors* for suggestions on retrying these failed sends.

exception anymail.exceptions.**AnymailInvalidAddress**

New in version 0.7.

The send call will raise a AnymailInvalidAddress error if you attempt to send a message with invalidly-formatted email addresses in the from_email or recipient lists.

One source of this error can be using a display-name ("real name") containing commas or parentheses. Per **RFC 5322**, you should use double quotes around the display-name portion of an email address:

```
# won't work:
send_mail(from_email='Widgets, Inc. <widgets@example.com>', ...)
# must use double quotes around display-name containing comma:
send_mail(from_email='"Widgets, Inc." <widgets@example.com>', ...)
```

exception anymail.exceptions.**AnymailSerializationError**

The send call will raise a AnymailSerializationError if there are message attributes Anymail doesn't know how to represent to your ESP.

The most common cause of this error is including values other than strings and numbers in your merge_data or metadata. (E.g., you need to format Decimal and date data to strings before setting them into merge_data.)

See *Formatting merge data* for more information.

1.4 Receiving mail

New in version 1.3.

For ESPs that support receiving inbound email, Anymail offers normalized handling of inbound events.

If you didn't set up webhooks when first installing Anymail, you'll need to *configure webhooks* to get started with inbound email. (You should also review *Securing webhooks*.)

Once you've enabled webhooks, Anymail will send a `anymail.signals.inbound` custom Django `signal` for each ESP inbound message it receives. You can connect your own receiver function to this signal for further processing. (This is very much like how Anymail handles *status tracking* events for sent messages. Inbound events just use a different signal receiver and have different event parameters.)

Be sure to read Django's listening to signals docs for information on defining and connecting signal receivers.

Example:

```
from anymail.signals import inbound
from django.dispatch import receiver

@receiver(inbound)  # add weak=False if inside some other function/class
def handle_inbound(sender, event, esp_name, **kwargs):
    message = event.message
    print("Received message from %s (envelope sender %s) with subject '%s'" % (
          message.from_email, message.envelope_sender, message.subject))
```

Some ESPs batch up multiple inbound messages into a single webhook call. Anymail will invoke your signal receiver once, separately, for each message in the batch.

Warning: Be careful with inbound email

Inbound email is user-supplied content. There are all kinds of ways a malicious sender can abuse the email format to give your app misleading or dangerous data. Treat inbound email content with the same suspicion you'd apply to any user-submitted data. Among other concerns:

- Senders can spoof the From header. An inbound message's `from_email` may or may not match the actual address that sent the message. (There are both legitimate and malicious uses for this capability.)
- Most other fields in email can be falsified. E.g., an inbound message's `date` may or may not accurately reflect when the message was sent.
- Inbound attachments have the same security concerns as user-uploaded files. If you process inbound attachments, you'll need to verify that the attachment content is valid.

  This is particularly important if you publish the attachment content through your app. For example, an "image" attachment could actually contain an executable file or raw HTML. You wouldn't want to serve that as a user's avatar.

  It's *not* sufficient to check the attachment's content-type or filename extension—senders can falsify both of those. Consider using python-magic or a similar approach to validate the *actual attachment content*.

The Django docs have additional notes on user-supplied content security.

1.4.1 Normalized inbound event

class anymail.signals.**AnymailInboundEvent**

The `event` parameter to Anymail's `inbound` *signal receiver* is an object with the following attributes:

message
An *AnymailInboundMessage* representing the email that was received. Most of what you're interested in will be on this *message* attribute. See the full details *below*.

event_type
A normalized str identifying the type of event. For inbound events, this is always 'inbound'.

timestamp
A datetime indicating when the inbound event was generated by the ESP, if available; otherwise None. (Very few ESPs provide this info.)

This is typically when the ESP received the message or shortly thereafter. (Use *event.message.date* if you're interested in when the message was sent.)

(The timestamp's timezone is often UTC, but the exact behavior depends on your ESP and account settings. Anymail ensures that this value is an *aware* datetime with an accurate timezone.)

event_id
A str unique identifier for the event, if available; otherwise None. Can be used to avoid processing the same event twice. The exact format varies by ESP, and very few ESPs provide an event_id for inbound messages.

An alternative approach to avoiding duplicate processing is to use the inbound message's *Message-ID* header (event.message['Message-ID']).

esp_event
The "raw" event data from the ESP, deserialized into a python data structure. For most ESPs this is either parsed JSON (as a dict), or sometimes the complete Django HttpRequest received by the webhook.

This gives you (non-portable) access to original event provided by your ESP, which can be helpful if you need to access data Anymail doesn't normalize.

1.4.2 Normalized inbound message

class anymail.inbound.**AnymailInboundMessage**
The message attribute of an AnymailInboundEvent is an AnymailInboundMessage—an extension of Python's standard email.message.Message with additional features to simplify inbound handling.

In addition to the base Message functionality, it includes these attributes:

envelope_sender
The actual sending address of the inbound message, as determined by your ESP. This is a str "addr-spec"—just the email address portion without any display name ("sender@example.com")—or None if the ESP didn't provide a value.

The envelope sender often won't match the message's From header—for example, messages sent on someone's behalf (mailing lists, invitations) or when a spammer deliberately falsifies the From address.

envelope_recipient
The actual destination address the inbound message was delivered to. This is a str "addr-spec"—just the email address portion without any display name ("recipient@example.com")—or None if the ESP didn't provide a value.

The envelope recipient may not appear in the To or Cc recipient lists—for example, if your inbound address is bcc'd on a message.

from_email
The value of the message's From header. Anymail converts this to an EmailAddress object, which makes it easier to access the parsed address fields:

```
>>> str(message.from_email)  # the fully-formatted address
'"Dr. Justin Customer, CPA" <jcustomer@example.com>'
>>> message.from_email.addr_spec  # the "email" portion of the address
'jcustomer@example.com'
>>> message.from_email.display_name  # empty string if no display name
'Dr. Justin Customer, CPA'
>>> message.from_email.domain
'example.com'
>>> message.from_email.username
'jcustomer'
```

(This API is borrowed from Python 3.6's `email.headerregistry.Address`.)

If the message has an invalid or missing From header, this property will be `None`. Note that From headers can be misleading; see *envelope_sender*.

to
A `list` of of parsed `EmailAddress` objects from the To header, or an empty list if that header is missing or invalid. Each address in the list has the same properties as shown above for *from_email*.

See *envelope_recipient* if you need to know the actual inbound address that received the inbound message.

cc
A `list` of of parsed `EmailAddress` objects, like *to*, but from the Cc headers.

subject
The value of the message's Subject header, as a `str`, or `None` if there is no Subject header.

date
The value of the message's Date header, as a `datetime` object, or `None` if the Date header is missing or invalid. This attribute will almost always be an aware datetime (with a timezone); in rare cases it can be naive if the sending mailer indicated that it had no timezone information available.

The Date header is the sender's claim about when it sent the message, which isn't necessarily accurate. (If you need to know when the message was received at your ESP, that might be available in *event.timestamp*. If not, you'd need to parse the messages's *Received* headers, which can be non-trivial.)

text
The message's plaintext message body as a `str`, or `None` if the message doesn't include a plaintext body.

html
The message's HTML message body as a `str`, or `None` if the message doesn't include an HTML body.

attachments
A `list` of all (non-inline) attachments to the message, or an empty list if there are no attachments. See *Handling Inbound Attachments* below for the contents of each list item.

inline_attachments
A `dict` mapping inline Content-ID references to attachment content. Each key is an "unquoted" cid without angle brackets. E.g., if the *html* body contains `<img src="cid:abc123...">`, you could get that inline image using `message.inline_attachments["abc123..."]`.

The content of each attachment is described in *Handling Inbound Attachments* below.

spam_score
A `float` spam score (usually from SpamAssassin) if your ESP provides it; otherwise `None`. The range of values varies by ESP and spam-filtering configuration, so you may need to experiment to find a useful threshold.

spam_detected
: If your ESP provides a simple yes/no spam determination, a `bool` indicating whether the ESP thinks the inbound message is probably spam. Otherwise `None`. (Most ESPs just assign a *`spam_score`* and leave its interpretation up to you.)

stripped_text
: If provided by your ESP, a simplified version the inbound message's plaintext body; otherwise `None`.

  What exactly gets "stripped" varies by ESP, but it often omits quoted replies and sometimes signature blocks. (And ESPs who do offer stripped bodies usually consider the feature experimental.)

stripped_html
: Like *`stripped_text`*, but for the HTML body. (Very few ESPs support this.)

Other headers, complex messages, etc.

You can use all of Python's `email.message.Message` features with an AnymailInboundMessage. For example, you can access message headers using Message's `mapping interface`:

```
message['reply-to']  # the Reply-To header (header keys are case-insensitive)
message.getall('DKIM-Signature')  # list of all DKIM-Signature headers
```

And you can use Message methods like `walk()` and `get_content_type()` to examine more-complex multipart MIME messages (digests, delivery reports, or whatever).

1.4.3 Handling Inbound Attachments

Anymail converts each inbound attachment to a specialized MIME object with additional methods for handling attachments and integrating with Django. It also backports some helpful MIME methods from newer versions of Python to all versions supported by Anymail.

The attachment objects in an AnymailInboundMessage's `attachments` list and `inline_attachments` dict have these methods:

class AnymailInboundMessage

as_uploaded_file()
: Returns the attachment converted to a Django `UploadedFile` object. This is suitable for assigning to a model's `FileField` or `ImageField`:

  ```
  # allow users to mail in jpeg attachments to set their profile avatars...
  if attachment.get_content_type() == "image/jpeg":
      # for security, you must verify the content is really a jpeg
      # (you'll need to supply the is_valid_jpeg function)
      if is_valid_jpeg(attachment.get_content_bytes()):
          user.profile.avatar_image = attachment.as_uploaded_file()
  ```

  See Django's docs on Managing files for more information on working with uploaded files.

get_content_type()

get_content_maintype()

get_content_subtype()
: The type of attachment content, as specified by the sender. (But remember attachments are essentially user-uploaded content, so you should *never trust the sender*.)

See the Python docs for more info on `email.message.Message.get_content_type()`, `get_content_maintype()`, and `get_content_subtype()`.

(Note that you *cannot* determine the attachment type using code like `issubclass(attachment, email.mime.image.MIMEImage)`. You should instead use something like `attachment.get_content_maintype() == 'image'`. The email package's specialized MIME subclasses are designed for constructing new messages, and aren't used for parsing existing, inbound email messages.)

get_filename()
The original filename of the attachment, as specified by the sender.

Never use this filename directly to write files—that would be a huge security hole. (What would your app do if the sender gave the filename "/etc/passwd" or "../settings.py"?)

is_attachment()
Returns `True` for a (non-inline) attachment, `False` otherwise. (Anymail back-ports Python 3.4.2's `is_attachment()` method to all supported versions.)

is_inline_attachment()
Returns `True` for an inline attachment (one with *Content-Disposition* "inline"), `False` otherwise.

get_content_disposition()
Returns the lowercased value (without parameters) of the attachment's *Content-Disposition* header. The return value should be either "inline" or "attachment", or `None` if the attachment is somehow missing that header.

(Anymail back-ports Python 3.5's `get_content_disposition()` method to all supported versions.)

get_content_text(*charset=None*, *errors='replace'*)
Returns the content of the attachment decoded to Unicode text. (This is generally only appropriate for text or message-type attachments.)

If provided, charset will override the attachment's declared charset. (This can be useful if you know the attachment's *Content-Type* has a missing or incorrect charset.)

The errors param is as in `decode()`. The default "replace" substitutes the Unicode "replacement character" for any illegal characters in the text.

Changed in version 2.1: Changed to use attachment's declared charset by default, and added errors option defaulting to replace.

get_content_bytes()
Returns the raw content of the attachment as bytes. (This will automatically decode any base64-encoded attachment data.)

Complex attachments

An Anymail inbound attachment is actually just an *AnymailInboundMessage* instance, following the Python email package's usual recursive representation of MIME messages. All *AnymailInboundMessage* and `email.message.Message` functionality is available on attachment objects (though of course not all features are meaningful in all contexts).

This can be helpful for, e.g., parsing email messages that are forwarded as attachments to an inbound message.

Anymail loads all attachment content into memory as it processes each inbound message. This may limit the size of attachments your app can handle, beyond any attachment size limits imposed by your ESP. Depending on how your ESP transmits attachments, you may also need to adjust Django's `DATA_UPLOAD_MAX_MEMORY_SIZE` setting to successfully receive larger attachments.

1.4.4 Inbound signal receiver functions

Your Anymail inbound signal receiver must be a function with this signature:

def my_handler(sender, event, esp_name, **kwargs):
(You can name it anything you want.)

Parameters

- **sender** (*class*) – The source of the event. (One of the `anymail.webhook.*` View classes, but you generally won't examine this parameter; it's required by Django's signal mechanism.)
- **event** (`AnymailInboundEvent`) – The normalized inbound event. Almost anything you'd be interested in will be in here—usually in the *`AnymailInboundMessage`* found in `event.message`.
- **esp_name** (*str*) – e.g., "SendMail" or "Postmark". If you are working with multiple ESPs, you can use this to distinguish ESP-specific handling in your shared event processing.
- ****kwargs** – Required by Django's signal mechanism (to support future extensions).

Returns nothing

Raises any exceptions in your signal receiver will result in a 400 HTTP error to the webhook. See discussion below.

If (any of) your signal receivers raise an exception, Anymail will discontinue processing the current batch of events and return an HTTP 400 error to the ESP. Most ESPs respond to this by re-sending the event(s) later, a limited number of times.

This is the desired behavior for transient problems (e.g., your Django database being unavailable), but can cause confusion in other error cases. You may want to catch some (or all) exceptions in your signal receiver, log the problem for later follow up, and allow Anymail to return the normal 200 success response to your ESP.

Some ESPs impose strict time limits on webhooks, and will consider them failed if they don't respond within (say) five seconds. And they may then retry sending these "failed" events, which could cause duplicate processing in your code. If your signal receiver code might be slow, you should instead queue the event for later, asynchronous processing (e.g., using something like Celery).

If your signal receiver function is defined within some other function or instance method, you *must* use the `weak=False` option when connecting it. Otherwise, it might seem to work at first, but will unpredictably stop being called at some point—typically on your production server, in a hard-to-debug way. See Django's docs on signals for more information.

1.5 Supported ESPs

Anymail currently supports these Email Service Providers. Click an ESP's name for specific Anymail settings required, and notes about any quirks or limitations:

1.5.1 Amazon SES

Anymail integrates with Amazon Simple Email Service (SES) using the Boto 3 AWS SDK for Python, and includes sending, tracking, and inbound receiving capabilities.

Alternatives

At least two other packages offer Django integration with Amazon SES: django-amazon-ses and django-ses. Depending on your needs, one of them may be more appropriate than Anymail.

New in version 2.1.

Installation

You must ensure the boto3 package is installed to use Anymail's Amazon SES backend. Either include the "amazon_ses" option when you install Anymail:

```
$ pip install django-anymail[amazon_ses]
```

or separately run `pip install boto3`.

To send mail with Anymail's Amazon SES backend, set:

```
EMAIL_BACKEND = "anymail.backends.amazon_ses.EmailBackend"
```

in your settings.py.

In addition, you must make sure boto3 is configured with AWS credentials having the necessary *IAM permissions*. There are several ways to do this; see Credentials in the Boto docs for options. Usually, an IAM role for EC2 instances, standard Boto environment variables, or a shared AWS credentials file will be appropriate. For more complex cases, use Anymail's *AMAZON_SES_CLIENT_PARAMS* setting to customize the Boto session.

Limitations and quirks

Hard throttling Like most ESPs, Amazon SES throttles sending for new customers. But unlike most ESPs, SES does not queue and slowly release throttled messages. Instead, it hard-fails the send API call. A strategy for *retrying errors* is required with any ESP; you're likely to run into it right away with Amazon SES.

Tags limitations Amazon SES's handling for tags is a bit different from other ESPs. Anymail tries to provide a useful, portable default behavior for its *tags* feature. See *Tags and metadata* below for more information and additional options.

No merge_metadata Amazon SES's batch sending API does not support the custom headers Anymail uses for metadata, so Anymail's *merge_metadata* feature is not available. (See *Tags and metadata* below for more information.)

Open and click tracking overrides Anymail's *track_opens* and *track_clicks* are not supported. Although Amazon SES *does* support open and click tracking, it doesn't offer a simple mechanism to override the settings for individual messages. If you need this feature, provide a custom ConfigurationSetName in Anymail's *esp_extra*.

No delayed sending Amazon SES does not support *send_at*.

No global send defaults for non-Anymail options With the Amazon SES backend, Anymail's *global send defaults* are only supported for Anymail's added message options (like *metadata* and *esp_extra*), not for standard EmailMessage attributes like `bcc` or `from_email`.

Arbitrary alternative parts allowed Amazon SES is one of the few ESPs that *does* support sending arbitrary alternative message parts (beyond just a single text/plain and text/html part).

Spoofed To header and multiple From emails allowed Amazon SES is one of the few ESPs that supports spoofing the *To* header (see *Additional headers*) and supplying multiple addresses in a message's `from_email`. (Most

ISPs consider these to be very strong spam signals, and using either them will almost certainly prevent delivery of your mail.)

Template limitations Messages sent with templates have a number of additional limitations, such as not supporting attachments. See *Batch sending/merge and ESP templates* below.

Tags and metadata

Amazon SES provides two mechanisms for associating additional data with sent messages, which Anymail uses to implement its `tags` and `metadata` features:

- **SES Message Tags** can be used for filtering or segmenting CloudWatch metrics and dashboards, and are available to Kinesis Firehose streams. (See "How do message tags work?" in the Amazon blog post Introducing Sending Metrics.)

  By default, Anymail does *not* use SES Message Tags. They have strict limitations on characters allowed, and are not consistently available to tracking webhooks. (They may be included in SES Event Publishing but not SES Notifications.)

- **Custom Email Headers** are available to all SNS notifications (webhooks), but not to CloudWatch or Kinesis.

  These are ordinary extension headers included in the sent message (and visible to recipients who view the full headers). There are no restrictions on characters allowed.

By default, Anymail uses only custom email headers. A message's `metadata` is sent JSON-encoded in a custom `X-Metadata` header, and a message's `tags` are sent in custom `X-Tag` headers. Both are available in Anymail's *tracking webhooks*.

Because Anymail `tags` are often used for segmenting reports, Anymail has an option to easily send an Anymail tag as an SES Message Tag that can be used in CloudWatch. Set the Anymail setting `AMAZON_SES_MESSAGE_TAG_NAME` to the name of an SES Message Tag whose value will be the *single* Anymail tag on the message. For example, with this setting:

```
ANYMAIL = {
    ...
    "AMAZON_SES_MESSAGE_TAG_NAME": "Type",
}
```

this send will appear in CloudWatch with the SES Message Tag `"Type": "Marketing"`:

```
message = EmailMessage(...)
message.tags = ["Marketing"]
message.send()
```

Anymail's `AMAZON_SES_MESSAGE_TAG_NAME` setting is disabled by default. If you use it, then only a single tag is supported, and both the tag and the name must be limited to alphanumeric, hyphen, and underscore characters.

For more complex use cases, set the SES `Tags` parameter directly in Anymail's *esp_extra*. See the example below. (Because custom headers do not work with SES's SendBulkTemplatedEmail call, esp_extra Tags is the only way to attach data to SES messages also using Anymail's `template_id` and `merge_data` features, and the `merge_metadata` cannot be supported.)

esp_extra support

To use Amazon SES features not directly supported by Anymail, you can set a message's `esp_extra` to a `dict` that will be merged into the params for the SendRawEmail or SendBulkTemplatedEmail SES API call.

Example:

```
message.esp_extra = {
    # Override AMAZON_SES_CONFIGURATION_SET_NAME for this message
    'ConfigurationSetName': 'NoOpenOrClickTrackingConfigSet',
    # Authorize a custom sender
    'SourceArn': 'arn:aws:ses:us-east-1:123456789012:identity/example.com',
    # Set Amazon SES Message Tags
    'Tags': [
        # (Names and values must be A-Z a-z 0-9 - and _ only)
        {'Name': 'UserID', 'Value': str(user_id)},
        {'Name': 'TestVariation', 'Value': 'Subject-Emoji-Trial-A'},
    ],
}
```

(You can also set `"esp_extra"` in Anymail's *global send defaults* to apply it to all messages.)

Batch sending/merge and ESP templates

Amazon SES offers *ESP stored templates* and *batch sending* with per-recipient merge data. See Amazon's Sending personalized email guide for more information.

When you set a message's *`template_id`* to the name of one of your SES templates, Anymail will use the SES SendBulkTemplatedEmail call to send template messages personalized with data from Anymail's normalized *`merge_data`* and *`merge_global_data`* message attributes.

```
message = EmailMessage(
    from_email="shipping@example.com",
    # you must omit subject and body (or set to None) with Amazon SES␣
↪templates
    to=["alice@example.com", "Bob <bob@example.com>"]
)
message.template_id = "MyTemplateName"  # Amazon SES TemplateName
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15",
}
```

Amazon's templated email APIs don't support several features available for regular email. When *`template_id`* is used:

- Attachments are not supported
- Extra headers are not supported
- Overriding the template's subject or body is not supported
- Anymail's *`metadata`* is not supported
- Anymail's *`tags`* are only supported with the *`AMAZON_SES_MESSAGE_TAG_NAME`* setting; only a single tag is allowed, and the tag is not directly available to webhooks. (See *Tags and metadata* above.)

Status tracking webhooks

Anymail can provide normalized *status tracking* notifications for messages sent through Amazon SES. SES offers two (confusingly) similar kinds of tracking, and Anymail supports both:

- SES Notifications include delivered, bounced, and complained (spam) Anymail *event_type*s. (Enabling these notifications may allow you to disable SES "email feedback forwarding.")
- SES Event Publishing also includes delivered, bounced and complained events, as well as sent, rejected, opened, clicked, and (template rendering) failed.

Both types of tracking events are delivered to Anymail's webhook URL through Amazon Simple Notification Service (SNS) subscriptions.

Amazon's naming here can be really confusing. We'll try to be clear about "SES Notifications" vs. "SES Event Publishing" as the two different kinds of SES tracking events. And then distinguish all of that from "SNS"—the publish/subscribe service used to notify Anymail's tracking webhooks about *both* kinds of SES tracking event.

To use Anymail's status tracking webhooks with Amazon SES:

1. First, *configure Anymail webhooks* and deploy your Django project. (Deploying allows Anymail to confirm the SNS subscription for you in step 3.)

Then in Amazon's **Simple Notification Service** console:

2. Create an SNS Topic to receive Amazon SES tracking events. The exact topic name is up to you; choose something meaningful like *SES_Tracking_Events*.
3. Subscribe Anymail's tracking webhook to the SNS Topic you just created. In the SNS console, click into the topic from step 2, then click the "Create subscription" button. For protocol choose HTTPS. For endpoint enter:

   `https://random:random@yoursite.example.com/anymail/amazon_ses/tracking/`

   - *random:random* is an *ANYMAIL_WEBHOOK_SECRET* shared secret
   - *yoursite.example.com* is your Django site

   Anymail will automatically confirm the SNS subscription. (For other options, see *Confirming SNS subscriptions* below.)

Finally, switch to Amazon's **Simple Email Service** console:

4. **If you want to use SES Notifications:** Follow Amazon's guide to configure SES notifications through SNS, using the SNS Topic you created above. Choose any event types you want to receive. Be sure to choose "Include original headers" if you need access to Anymail's *metadata* or *tags* in your webhook handlers.
5. **If you want to use SES Event Publishing:**
   a. Follow Amazon's guide to create an SES "Configuration Set". Name it something meaningful, like *TrackingConfigSet*.
   b. Follow Amazon's guide to add an SNS event destination for SES event publishing, using the SNS Topic you created above. Choose any event types you want to receive.
   c. Update your Anymail settings to send using this Configuration Set by default:

```
ANYMAIL = {
    ...
    "AMAZON_SES_CONFIGURATION_SET_NAME": "TrackingConfigSet",
}
```

Caution: The delivery, bounce, and complaint event types are available in both SES Notifications *and* SES Event Publishing. If you're using both, don't enable the same events in both places, or you'll receive duplicate notifications with *different* *event_id*s.

Note that Amazon SES's open and click tracking does not distinguish individual recipients. If you send a single message to multiple recipients, Anymail will call your tracking handler with the "opened" or "clicked" event for *every*

original recipient of the message, including all to, cc and bcc addresses. (Amazon recommends avoiding multiple recipients with SES.)

In your tracking signal receiver, the normalized AnymailTrackingEvent's *esp_event* will be set to the the parsed, top-level JSON event object from SES: either SES Notification contents or SES Event Publishing contents. (The two formats are nearly identical.) You can use this to obtain SES Message Tags (see *Tags and metadata*) from SES Event Publishing notifications:

```
from anymail.signals import tracking
from django.dispatch import receiver

@receiver(tracking)  # add weak=False if inside some other function/class
def handle_tracking(sender, event, esp_name, **kwargs):
    if esp_name == "Amazon SES":
        try:
            message_tags = {
                name: values[0]
                for name, values in event.esp_event["mail"]["tags"].items()}
        except KeyError:
            message_tags = None  # SES Notification (not Event Publishing) event
        print("Message %s to %s event %s: Message Tags %r" % (
              event.message_id, event.recipient, event.event_type, message_tags))
```

Anymail does *not* currently check SNS signature verification, because Amazon has not released a standard way to do that in Python. Instead, Anymail relies on your *WEBHOOK_SECRET* to verify SNS notifications are from an authorized source.

Note: Amazon SNS's default policy for handling HTTPS notification failures is to retry three times, 20 seconds apart, and then drop the notification. That means **if your webhook is ever offline for more than one minute, you may miss events.**

For most uses, it probably makes sense to configure an SNS retry policy with more attempts over a longer period. E.g., 20 retries ranging from 5 seconds minimum to 600 seconds (5 minutes) maximum delay between attempts, with geometric backoff.

Also, SNS does *not* guarantee notifications will be delivered to HTTPS subscribers like Anymail webhooks. The longest SNS will ever keep retrying is one hour total. If you need retries longer than that, or guaranteed delivery, you may need to implement your own queuing mechanism with something like Celery or directly on Amazon Simple Queue Service (SQS).

Inbound webhook

You can receive email through Amazon SES with Anymail's normalized *inbound* handling. See Receiving email with Amazon SES for background.

Configuring Anymail's inbound webhook for Amazon SES is similar to installing the *tracking webhook*. You must use a different SNS Topic for inbound.

To use Anymail's inbound webhook with Amazon SES:

1. First, if you haven't already, *configure Anymail webhooks* and deploy your Django project. (Deploying allows Anymail to confirm the SNS subscription for you in step 3.)
2. Create an SNS Topic to receive Amazon SES inbound events. The exact topic name is up to you; choose something meaningful like *SES_Inbound_Events*. (If you are also using Anymail's tracking events, this must be a *different* SNS Topic.)

3. Subscribe Anymail's inbound webhook to the SNS Topic you just created. In the SNS console, click into the topic from step 2, then click the "Create subscription" button. For protocol choose HTTPS. For endpoint enter:

   `https://random:random@yoursite.example.com/anymail/amazon_ses/inbound/`

   - *random:random* is an `ANYMAIL_WEBHOOK_SECRET` shared secret
   - *yoursite.example.com* is your Django site

   Anymail will automatically confirm the SNS subscription. (For other options, see *Confirming SNS subscriptions* below.)

4. Next, follow Amazon's guide to Setting up Amazon SES email receiving. There are several steps. Come back here when you get to "Action Options" in the last step, "Creating Receipt Rules."

5. Anymail supports two SES receipt actions: S3 and SNS. (Both actually use SNS.) You can choose either one: the SNS action is easier to set up, but the S3 action allows you to receive larger messages and can be more robust. (You can change at any time, but don't use both simultaneously.)

   - **For the SNS action:** choose the SNS Topic you created in step 2. Anymail will handle either Base64 or UTF-8 encoding; use Base64 if you're not sure.
   - **For the S3 action:** choose or create any S3 bucket that Boto will be able to read. (See *IAM permissions*; *don't* use a world-readable bucket!) "Object key prefix" is optional. Anymail does *not* currently support the "Encrypt message" option. Finally, choose the SNS Topic you created in step 2.

Amazon SES will likely deliver a test message to your Anymail inbound handler immediately after you complete the last step.

If you are using the S3 receipt action, note that Anymail does not delete the S3 object. You can delete it from your code after successful processing, or set up S3 bucket policies to automatically delete older messages. In your inbound handler, you can retrieve the S3 object key by prepending the "object key prefix" (if any) from your receipt rule to Anymail's `event.event_id`.

Amazon SNS imposes a 15 second limit on all notifications. This includes time to download the message (if you are using the S3 receipt action) and any processing in your signal receiver. If the total takes longer, SNS will consider the notification failed and will make several repeat attempts. To avoid problems, it's essential any lengthy operations are offloaded to a background task.

Amazon SNS's default retry policy times out after one minute of failed notifications. If your webhook is ever unreachable for more than a minute, **you may miss inbound mail.** You'll probably want to adjust your SNS topic settings to reduce the chances of that. See the note about *retry policies* in the tracking webhooks discussion above.

In your inbound signal receiver, the normalized AnymailTrackingEvent's `esp_event` will be set to the the parsed, top-level JSON object described in SES Email Receiving contents.

Confirming SNS subscriptions

Amazon SNS requires HTTPS endpoints (webhooks) to confirm they actually want to subscribe to an SNS Topic. See Sending SNS messages to HTTPS endpoints in the Amazon SNS docs for more information.

(This has nothing to do with verifying email identities in Amazon *SES*, and is not related to email recipients confirming subscriptions to your content.)

Anymail will automatically handle SNS endpoint confirmation for you, for both tracking and inbound webhooks, if both:

1. You have deployed your Django project with *Anymail webhooks enabled* and an Anymail `WEBHOOK_SECRET` set, before subscribing the SNS Topic to the webhook URL.

   (If you subscribed the SNS topic too early, you can re-send the confirmation request later from the Subscriptions section of the Amazon SNS dashboard.)

2. The SNS endpoint URL includes the correct Anymail *WEBHOOK_SECRET* as HTTP basic authentication. (Amazon SNS only allows this with https urls, not plain http.)

   Anymail requires a valid secret to ensure the subscription request is coming from you, not some other AWS user.

If you do not want Anymail to automatically confirm SNS subscriptions for its webhook URLs, set *AMAZON_SES_AUTO_CONFIRM_SNS_SUBSCRIPTIONS* to `False` in your ANYMAIL settings.

When auto-confirmation is disabled (or if Anymail receives an unexpected confirmation request), it will raise an `AnymailWebhookValidationFailure`, which should show up in your Django error logging. The error message will include the Token you can use to manually confirm the subscription in the Amazon SNS dashboard or through the SNS API.

Settings

Additional Anymail settings for use with Amazon SES:

AMAZON_SES_CLIENT_PARAMS

Optional. Additional client parameters Anymail should use to create the boto3 session client. Example:

```
ANYMAIL = {
    ...
    "AMAZON_SES_CLIENT_PARAMS": {
        # example: override normal Boto credentials specifically for Anymail
        "aws_access_key_id": os.getenv("AWS_ACCESS_KEY_FOR_ANYMAIL_SES"),
        "aws_secret_access_key": os.getenv("AWS_SECRET_KEY_FOR_ANYMAIL_SES"),
        "region_name": "us-west-2",
        # override other default options
        "config": {
            "connect_timeout": 30,
            "read_timeout": 30,
        }
    },
}
```

In most cases, it's better to let Boto obtain its own credentials through one of its other mechanisms: an IAM role for EC2 instances, standard AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY and AWS_SESSION_TOKEN environment variables, or a shared AWS credentials file.

AMAZON_SES_SESSION_PARAMS

Optional. Additional session parameters Anymail should use to create the boto3 Session. Example:

```
ANYMAIL = {
    ...
    "AMAZON_SES_SESSION_PARAMS": {
        "profile_name": "anymail-testing",
    },
}
```

AMAZON_SES_CONFIGURATION_SET_NAME

Optional. The name of an Amazon SES Configuration Set Anymail should use when sending messages. The default is to send without any Configuration Set. Note that a Configuration Set is required to receive SES Event Publishing tracking events. See *Status tracking webhooks* above.

You can override this for individual messages with *esp_extra*.

AMAZON_SES_MESSAGE_TAG_NAME

Optional, default `None`. The name of an Amazon SES "Message Tag" whose value is set from a message's Anymail `tags`. See *Tags and metadata* above.

AMAZON_SES_AUTO_CONFIRM_SNS_SUBSCRIPTIONS

Optional boolean, default `True`. Set to `False` to prevent Anymail webhooks from automatically accepting Amazon SNS subscription confirmation requests. See *Confirming SNS subscriptions* above.

IAM permissions

Anymail requires IAM permissions that will allow it to use these actions:

- To send mail:
  - Ordinary (non-templated) sends: `ses:SendRawEmail`
  - Template/merge sends: `ses:SendBulkTemplatedEmail`
- To *automatically confirm* webhook SNS subscriptions: `sns:ConfirmSubscription`
- For status tracking webhooks: no special permissions
- To receive inbound mail:
  - With an "SNS action" receipt rule: no special permissions
  - With an "S3 action" receipt rule: `s3:GetObject` on the S3 bucket and prefix used (or S3 Access Control List read access for inbound messages in that bucket)

This IAM policy covers all of those:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["ses:SendRawEmail", "ses:SendBulkTemplatedEmail"],
    "Resource": "*"
  }, {
    "Effect": "Allow",
    "Action": ["sns:ConfirmSubscription"],
    "Resource": ["arn:aws:sns:*:*:*"]
  }, {
    "Effect": "Allow",
    "Action": ["s3:GetObject"],
    "Resource": ["arn:aws:s3:::MY-PRIVATE-BUCKET-NAME/MY-INBOUND-PREFIX/*"]
  }]
}
```

Following the principle of least privilege, you should omit permissions for any features you aren't using, and you may want to add additional restrictions:

- For Amazon SES sending, you can add conditions to restrict senders, recipients, times, or other properties. See Amazon's Controlling access to Amazon SES guide.
- For auto-confirming webhooks, you might limit the resource to SNS topics owned by your AWS account, and/or specific topic names or patterns. E.g., `"arn:aws:sns:*:0000000000000000:SES_*_Events"` (replacing the zeroes with your numeric AWS account id). See Amazon's guide to Amazon SNS ARNs.
- For inbound S3 delivery, there are multiple ways to control S3 access and data retention. See Amazon's Managing access permissions to your Amazon S3 resources. (And obviously, you should *never store incoming emails to a public bucket!*)

  Also, you may need to grant Amazon SES (but *not* Anymail) permission to *write* to your inbound bucket. See Amazon's Giving permissions to Amazon SES for email receiving.
- For all operations, you can limit source IP, allowable times, user agent, and more. (Requests from Anymail will include "django-anymail/*version*" along with Boto's user-agent.) See Amazon's guide to IAM condition context keys.

1.5.2 Mailgun

Anymail integrates with the Mailgun transactional email service from Rackspace, using their REST API.

Settings

EMAIL_BACKEND

To use Anymail's Mailgun backend, set:

```
EMAIL_BACKEND = "anymail.backends.mailgun.EmailBackend"
```

in your settings.py.

MAILGUN_API_KEY

Required for sending. Your Mailgun "Private API key" from the Mailgun API security settings:

```
ANYMAIL = {
    ...
    "MAILGUN_API_KEY": "<your API key>",
}
```

Anymail will also look for `MAILGUN_API_KEY` at the root of the settings file if neither `ANYMAIL["MAILGUN_API_KEY"]` nor `ANYMAIL_MAILGUN_API_KEY` is set.

MAILGUN_SENDER_DOMAIN

If you are using a specific Mailgun sender domain that is *different* from your messages' `from_email` domains, set this to the domain you've configured in your Mailgun account.

If your messages' `from_email` domains always match a configured Mailgun sender domain, this setting is not needed.

See *Email sender domain* below for examples.

MAILGUN_WEBHOOK_SIGNING_KEY

New in version 6.1.

Required for tracking or inbound webhooks. Your "HTTP webhook signing key" from the Mailgun API security settings:

```
ANYMAIL = {
    ...
    "MAILGUN_WEBHOOK_SIGNING_KEY": "<your webhook signing key>",
}
```

If not provided, Anymail will attempt to validate webhooks using the *MAILGUN_API_KEY* setting instead. (These two keys have the same values for new Mailgun users, but will diverge if you ever rotate either key.)

MAILGUN_API_URL

The base url for calling the Mailgun API. It does not include the sender domain. (Anymail *figures this out* for you.)

The default is `MAILGUN_API_URL = "https://api.mailgun.net/v3"`, which connects to Mailgun's US service. You must override this if you are using Mailgun's European region:

```
ANYMAIL = {
  "MAILGUN_API_KEY": "...",
  "MAILGUN_API_URL": "https://api.eu.mailgun.net/v3",
  # ...
}
```

Email sender domain

Mailgun's API requires identifying the sender domain. By default, Anymail uses the domain of each messages's `from_email` (e.g., "example.com" for "from@example.com").

You will need to override this default if you are using a dedicated Mailgun sender domain that is different from a message's `from_email` domain.

For example, if you are sending from "orders@example.com", but your Mailgun account is configured for "*mail1*.example.com", you should provide *MAILGUN_SENDER_DOMAIN* in your settings.py:

```
ANYMAIL = {
    ...
    "MAILGUN_API_KEY": "<your API key>",
    "MAILGUN_SENDER_DOMAIN": "mail1.example.com"
}
```

If you need to override the sender domain for an individual message, use Anymail's *envelope_sender* (only the domain is used; anything before the @ is ignored):

```
message = EmailMessage(from_email="marketing@example.com", ...)
message.envelope_sender = "anything@mail2.example.com"  # the "anything@" is␣
↪ignored
```

Changed in version 2.0: Earlier Anymail versions looked for a special `sender_domain` key in the message's *esp_extra* to override Mailgun's sender domain. This is still supported, but may be deprecated in a future release. Using *envelope_sender* as shown above is now preferred.

exp_extra support

Anymail's Mailgun backend will pass all *esp_extra* values directly to Mailgun. You can use any of the (non-file) parameters listed in the Mailgun sending docs. Example:

```
message = AnymailMessage(...)
message.esp_extra = {
    'o:testmode': 'yes',  # use Mailgun's test mode
}
```

Limitations and quirks

Attachments require filenames Mailgun has an undocumented API requirement that every attachment must have a filename. Attachments with missing filenames are silently dropped from the sent message. Similarly, every inline attachment must have a *Content-ID*.

To avoid unexpected behavior, Anymail will raise an *AnymailUnsupportedFeature* error if you attempt to send a message through Mailgun with any attachments that don't have filenames (or inline attachments that don't have *Content-ID*s).

Ensure your attachments have filenames by using `message.attach_file(filename)`, `message.attach(content, filename="...")`, or if you are constructing your own MIME objects to attach, `mimeobj.add_header("Content-Disposition", "attachment", filename="...")`.

Ensure your inline attachments have Content-IDs by using Anymail's *inline image helpers*, or if you are constructing your own MIME objects, `mimeobj.add_header("Content-ID", "...")` and `mimeobj.add_header("Content-Disposition", "inline")`.

Changed in version 4.3: Earlier Anymail releases did not check for these cases, and attachments without filenames/Content-IDs would be ignored by Mailgun without notice.

Envelope sender uses only domain Anymail's *envelope_sender* is used to select your Mailgun *sender domain*. For obvious reasons, only the domain portion applies. You can use anything before the @, and it will be ignored.

Using merge_metadata with merge_data If you use both Anymail's *merge_data* and *merge_metadata* features, make sure your merge_data keys do not start with `v:`. (It's a good idea anyway to avoid colons and other special characters in merge_data keys, as this isn't generally portable to other ESPs.)

The same underlying Mailgun feature ("recipient-variables") is used to implement both Anymail features. To avoid conflicts, Anymail prepends `v:` to recipient variables needed for merge_metadata. (This prefix is stripped as Mailgun prepares the message to send, so it won't be present in your Mailgun API logs or the metadata that is sent to tracking webhooks.)

Additional limitations on merge_data with template_id If you are using Mailgun's stored handlebars templates (Anymail's *template_id*), *merge_data* cannot contain complex types or have any keys that conflict with *metadata*. See *Limitations with stored handlebars templates* below for more details.

merge_metadata values default to empty string If you use Anymail's *merge_metadata* feature, and you supply metadata keys for some recipients but not others, Anymail will first try to resolve the missing keys in *metadata*, and if they are not found there will default them to an empty string value.

Your tracking webhooks will receive metadata values (either that you provided or the default empty string) for *every* key used with *any* recipient in the send.

Batch sending/merge and ESP templates

Mailgun supports *ESP stored templates*, on-the-fly templating, and *batch sending* with per-recipient merge data.

Changed in version 7.0: Added support for Mailgun's stored (handlebars) templates.

Mailgun has two different syntaxes for substituting data into templates:

- "Recipient variables" look like `%recipient.name%`, and are used with on-the-fly templates. You can refer to a recipient variable inside a message's body, subject, or other message attributes defined in your Django code. See Mailgun batch sending for more information. (Note that Mailgun's docs also sometimes refer to recipient variables as "template *variables*," and there are some additional predefined ones described in their docs.)
- "Template *substitutions*" look like `{{ name }}`, and can *only* be used in handlebars templates that are defined and stored in your Mailgun account (via the Mailgun dashboard or API). You refer to a stored template using Anymail's *`template_id`* in your Django code. See Mailgun templates for more information.

With either type of template, you supply the substitution data using Anymail's normalized *`merge_data`* and *`merge_global_data`* message attributes. Anymail will figure out the correct Mailgun API parameters to use.

Here's an example defining an on-the-fly template that uses Mailgun recipient variables:

```
message = EmailMessage(
    from_email="shipping@example.com",
    # Use %recipient.___% syntax in subject and body:
    subject="Your order %recipient.order_no% has shipped",
    body="""Hi %recipient.name%,
            We shipped your order %recipient.order_no%
            on %recipient.ship_date%.""",
    to=["alice@example.com", "Bob <bob@example.com>"]
)
# (you'd probably also set a similar html body with %recipient.___%␣
↪variables)
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15"  # Anymail maps globals to all recipients
}
```

And here's an example that uses the same data with a stored template, which could refer to `{{ name }}`, `{{ order_no }}`, and `{{ ship_date }}` in its definition:

```
message = EmailMessage(
    from_email="shipping@example.com",
    # The message body and html_body come from from the stored template.
    # (You can still use %recipient.___% fields in the subject:)
    subject="Your order %recipient.order_no% has shipped",
    to=["alice@example.com", "Bob <bob@example.com>"]
)
message.template_id = 'shipping-notification'  # name of template in our␣
↪account
# The substitution data is exactly the same as in the previous example:
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15"  # Anymail maps globals to all recipients
}
```

When you supply per-recipient *`merge_data`*, Anymail supplies Mailgun's `recipient-variables` parameter, which puts Mailgun in batch sending mode so that each "to" recipient sees only their own email address. (Any cc's or

bcc's will be duplicated for *every* to-recipient.)

If you want to use batch sending with a regular message (without a template), set merge data to an empty dict: `message.merge_data = {}`.

Mailgun does not natively support global merge data. Anymail emulates the capability by copying any *merge_global_data* values to every recipient.

Limitations with stored handlebars templates

Although Anymail tries to insulate you from Mailgun's relatively complicated API parameters for template substitutions in batch sends, there are two cases it can't handle. These *only* apply to stored handlebars templates (when you've set Anymail's *template_id* attribute).

First, metadata and template merge data substitutions use the same underlying "custom data" API parameters when a handlebars template is used. If you have any duplicate keys between your tracking metadata (*metadata*/*merge_metadata*) and your template merge data (*merge_data*/*merge_global_data*), Anymail will raise an *AnymailUnsupportedFeature* error.

Second, Mailgun's API does not allow complex data types like lists or dicts to be passed as template substitutions for a batch send (confirmed with Mailgun support 8/2019). Your Anymail *merge_data* and *merge_global_data* should only use simple types like string or number. This means you cannot use the handlebars `{{#each item}}` block helper or dotted field notation like `{{object.field}}` with data passed through Anymail's normalized merge data attributes.

Most ESPs do not support complex merge data types, so trying to do that is not recommended anyway, for portability reasons. But if you *do* want to pass complex types to Mailgun handlebars templates, and you're only sending to one recipient at a time, here's a (non-portable!) workaround:

```
# Using complex substitutions with Mailgun handlebars templates.
# This works only for a single recipient, and is not at all portable between␣
↪ESPs.
message = EmailMessage(
    from_email="shipping@example.com",
    to=["alice@example.com"]  # single recipient *only* (no batch send)
    subject="Your order has shipped",  # recipient variables *not* available
)
message.template_id = 'shipping-notification'  # name of template in our␣
↪account
substitutions = {
    'items': [  # complex substitution data
        {'product': "Anvil", 'quantity': 1},
        {'product': "Tacks", 'quantity': 100},
    ],
    'ship_date': "May 15",
}
# Do *not* set Anymail's message.merge_data, merge_global_data, or merge_
↪metadata.
# Instead add Mailgun custom variables directly:
message.extra_headers['X-Mailgun-Variables'] = json.dumps(substitutions)
```

Status tracking webhooks

Changed in version 4.0: Added support for Mailgun's June, 2018 (non-"legacy") webhook format.

Changed in version 6.1: Added support for a new *MAILGUN_WEBHOOK_SIGNING_KEY* setting, separate from your MAILGUN_API_KEY.

If you are using Anymail's normalized *status tracking*, enter the url in the Mailgun webhooks config for your domain. (Be sure to select the correct sending domain—Mailgun's sandbox and production domains have separate webhook settings.)

Mailgun allows you to enter a different URL for each event type: just enter this same Anymail tracking URL for all events you want to receive:

`https://random:random@yoursite.example.com/anymail/mailgun/tracking/`

- *random:random* is an *ANYMAIL_WEBHOOK_SECRET* shared secret
- *yoursite.example.com* is your Django site

Mailgun implements a limited form of webhook signing, and Anymail will verify these signatures against your *MAILGUN_WEBHOOK_SIGNING_KEY* Anymail setting. By default, Mailgun's webhook signature provides similar security to Anymail's shared webhook secret, so it's acceptable to omit the *ANYMAIL_WEBHOOK_SECRET* setting (and "{random}:{random}@" portion of the webhook url) with Mailgun webhooks.

Mailgun will report these Anymail *event_type*s: delivered, rejected, bounced, complained, unsubscribed, opened, clicked.

The event's *esp_event* field will be the parsed Mailgun webhook payload as a Python `dict` with `"signature"` and `"event-data"` keys.

Anymail uses Mailgun's webhook `token` as its normalized *event_id*, rather than Mailgun's event-data `id` (which is only guaranteed to be unique during a single day). If you need the event-data id, it can be accessed in your webhook handler as `event.esp_event["event-data"]["id"]`. (This can be helpful for working with Mailgun's other event APIs.)

Note: Mailgun legacy webhooks

In late June, 2018, Mailgun introduced a new set of webhooks with an improved payload design, and at the same time renamed their original webhooks to "Legacy Webhooks."

Anymail v4.0 and later supports both new and legacy Mailgun webhooks, and the same Anymail webhook url works as either. Earlier Anymail versions can only be used as legacy webhook urls.

The new (non-legacy) webhooks are preferred, particularly with Anymail's *metadata* and *tags* features. But if you have already configured the legacy webhooks, there is no need to change.

If you are using Mailgun's legacy webhooks:

- The *event.esp_event* field will be a Django `QueryDict` of Mailgun event fields (the raw POST data provided by legacy webhooks).
- You should avoid using "body-plain," "h," "message-headers," "message-id" or "tag" as *metadata* keys. A design limitation in Mailgun's legacy webhooks prevents Anymail from reliably retrieving this metadata from opened, clicked, and unsubscribed events. (This is not an issue with the newer, non-legacy webhooks.)

Inbound webhook

If you want to receive email from Mailgun through Anymail's normalized *inbound* handling, follow Mailgun's Receiving, Storing and Fowarding Messages guide to set up an inbound route that forwards to Anymail's inbound webhook. (You can configure routes using Mailgun's API, or simply using the Mailgun receiving config.)

The *action* for your route will be either:

```
forward("https://random:random@yoursite.example.com/anymail/mailgun/
inbound/")        forward("https://random:random@yoursite.example.com/
anymail/mailgun/inbound_mime/")
```

- *random:random* is an *ANYMAIL_WEBHOOK_SECRET* shared secret
- *yoursite.example.com* is your Django site

Anymail accepts either of Mailgun's "fully-parsed" (.../inbound/) and "raw MIME" (.../inbound_mime/) formats; the URL tells Mailgun which you want. Because Anymail handles parsing and normalizing the data, both are equally easy to use. The raw MIME option will give the most accurate representation of *any* received email (including complex forms like multi-message mailing list digests). The fully-parsed option *may* use less memory while processing messages with many large attachments.

If you want to use Anymail's normalized *spam_detected* and *spam_score* attributes, you'll need to set your Mailgun domain's inbound spam filter to "Deliver spam, but add X-Mailgun-SFlag and X-Mailgun-SScore headers" (in the Mailgun domains config).

Anymail will verify Mailgun inbound message events using your *MAILGUN_WEBHOOK_SIGNING_KEY* Anymail setting. By default, Mailgun's webhook signature provides similar security to Anymail's shared webhook secret, so it's acceptable to omit the *ANYMAIL_WEBHOOK_SECRET* setting (and "{random}:{random}@" portion of the action) with Mailgun inbound routing.

1.5.3 Mailjet

Anymail integrates with the Mailjet email service, using their transactional Send API (v3).

New in version 0.11.

Note: Mailjet has released a newer v3.1 Send API, but due to mismatches between its documentation and actual behavior, Anymail has been unable to switch to it. Anymail's maintainers have reported the problems to Mailjet, and if and when they are resolved, Anymail will switch to the v3.1 API. This change should be largely transparent to your code, unless you are using Anymail's *esp_extra* feature to set API-specific options.

Settings

EMAIL_BACKEND

To use Anymail's Mailjet backend, set:

```
EMAIL_BACKEND = "anymail.backends.mailjet.EmailBackend"
```

in your settings.py.

MAILJET_API_KEY and MAILJET_SECRET_KEY

Your Mailjet API key and secret key, from your Mailjet account REST API settings under API Key Management. (Mailjet's documentation also sometimes uses "API private key" to mean the same thing as "secret key.")

```
ANYMAIL = {
    ...
    "MAILJET_API_KEY": "<your API key>",
    "MAILJET_SECRET_KEY": "<your API secret>",
}
```

You can use either a master or sub-account API key.

Anymail will also look for `MAILJET_API_KEY` and `MAILJET_SECRET_KEY` at the root of the settings file if neither `ANYMAIL["MAILJET_API_KEY"]` nor `ANYMAIL_MAILJET_API_KEY` is set.

MAILJET_API_URL

The base url for calling the Mailjet API.

The default is `MAILJET_API_URL = "https://api.mailjet.com/v3"` (It's unlikely you would need to change this. This setting cannot be used to opt into a newer API version; the parameters are not backwards compatible.)

esp_extra support

To use Mailjet features not directly supported by Anymail, you can set a message's *esp_extra* to a `dict` of Mailjet's Send API json properties. Your `esp_extra` dict will be merged into the parameters Anymail has constructed for the send, with `esp_extra` having precedence in conflicts.

Note: Any `esp_extra` settings will need to be updated when Anymail changes to use Mailjet's upcoming v3.1 API. (See *note above*.)

Example:

```
message.esp_extra = {
    # Mailjet v3.0 Send API options:
    "Mj-prio": 3,  # Use Mailjet critically-high priority queue
    "Mj-CustomID": my_event_tracking_id,
}
```

(You can also set `"esp_extra"` in Anymail's *global send defaults* to apply it to all messages.)

Limitations and quirks

Single tag Anymail uses Mailjet's campaign option for tags, and Mailjet allows only a single campaign per message. If your message has two or more *tags*, you'll get an *AnymailUnsupportedFeature* error—or if you've enabled *ANYMAIL_IGNORE_UNSUPPORTED_FEATURES*, Anymail will use only the first tag.

No delayed sending Mailjet does not support *send_at*.

Envelope sender may require approval Anymail passes *envelope_sender* to Mailjet, but this may result in an API error if you have not received special approval from Mailjet support to use custom senders.

Commas in recipient names Mailjet's v3 API does not properly handle commas in recipient display-names. (Tested July, 2017, and confirmed with Mailjet API support.)

If your message would be affected, Anymail attempts to work around the problem by switching to MIME encoded-word syntax where needed.

Most modern email clients should support this syntax, but if you run into issues, you might want to strip commas from all recipient names (in `to`, `cc`, *and* `bcc`) before sending.

(This should be resolved in a future release when Anymail *switches* to Mailjet's upcoming v3.1 API.)

Changed in version 6.0: Earlier versions of Anymail were unable to mix `cc` or `bcc` fields and *merge_data* in the same Mailjet message. This limitation was removed in Anymail 6.0.

Batch sending/merge and ESP templates

Mailjet offers both *ESP stored templates* and *batch sending* with per-recipient merge data.

You can use a Mailjet stored transactional template by setting a message's *template_id* to the template's *numeric* template ID. (*Not* the template's name. To get the numeric template id, click on the name in your Mailjet transactional templates, then look for "Template ID" above the preview that appears.)

Supply the template merge data values with Anymail's normalized *merge_data* and *merge_global_data* message attributes.

```
message = EmailMessage(
    ...
    # omit subject and body (or set to None) to use template content
    to=["alice@example.com", "Bob <bob@example.com>"]
)
message.template_id = "176375"  # Mailjet numeric template id
message.from_email = None  # Use the From address stored with the template
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15",
}
```

Any `from_email` in your EmailMessage will override the template's default sender address. To use the template's sender, you must explicitly set `from_email = None` after creating the EmailMessage, as shown above. (If you omit this, Django's default `DEFAULT_FROM_EMAIL` will be used.)

Instead of creating a stored template at Mailjet, you can also refer to merge fields directly in an EmailMessage's body—the message itself is used as an on-the-fly template:

```
message = EmailMessage(
    from_email="orders@example.com",
    to=["alice@example.com", "Bob <bob@example.com>"],
    subject="Your order has shipped",  # subject doesn't support on-the-fly␣
↪merge fields
    # Use [[var:FIELD]] to for on-the-fly merge into plaintext or html body:
    body="Dear [[var:name]]: Your order [[var:order_no]] shipped on␣
↪[[var:ship_date]]."
)
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15",
}
```

(Note that on-the-fly templates use square brackets to indicate "personalization" merge fields, rather than the curly brackets used with stored templates in Mailjet's template language.)

See Mailjet's template documentation and template language docs for more information.

Status tracking webhooks

If you are using Anymail's normalized *status tracking*, enter the url in your Mailjet account REST API settings under Event tracking (triggers):

`https://random:random@yoursite.example.com/anymail/mailjet/tracking/`

- *random:random* is an `ANYMAIL_WEBHOOK_SECRET` shared secret
- *yoursite.example.com* is your Django site

Be sure to enter the URL in the Mailjet settings for all the event types you want to receive. It's also recommended to select the "group events" checkbox for each trigger, to minimize your server load.

Mailjet will report these Anymail `event_type`s: rejected, bounced, deferred, delivered, opened, clicked, complained, unsubscribed.

The event's `esp_event` field will be a `dict` of Mailjet event fields, for a single event. (Although Mailjet calls webhooks with batches of events, Anymail will invoke your signal receiver separately for each event in the batch.)

Inbound webhook

If you want to receive email from Mailjet through Anymail's normalized *inbound* handling, follow Mailjet's Parse API inbound emails guide to set up Anymail's inbound webhook.

The parseroute Url parameter will be:

`https://random:random@yoursite.example.com/anymail/mailjet/inbound/`

- *random:random* is an `ANYMAIL_WEBHOOK_SECRET` shared secret
- *yoursite.example.com* is your Django site

Once you've done Mailjet's "basic setup" to configure the Parse API webhook, you can skip ahead to the "use your own domain" section of their guide. (Anymail normalizes the inbound event for you, so you won't need to worry about Mailjet's event and attachment formats.)

1.5.4 Mandrill

Anymail integrates with the Mandrill transactional email service from MailChimp.

Note: Limited Support for Mandrill

Anymail is developed to the public Mandrill documentation, but unlike other supported ESPs, we are unable to test or debug against the live Mandrill APIs. (MailChimp discourages use of Mandrill by "developers," and doesn't offer testing access for packages like Anymail.)

As a result, Anymail bugs with Mandrill will generally be discovered by Anymail's users, in production; Anymail's maintainers often won't be able to answer Mandrill-specific questions; and fixes and improvements for Mandrill will tend to lag other ESPs.

If you are integrating only Mandrill, and not considering one of Anymail's other ESPs, you might prefer using MailChimp's official mandrill python package instead of Anymail.

Settings

EMAIL_BACKEND

To use Anymail's Mandrill backend, set:

```
EMAIL_BACKEND = "anymail.backends.mandrill.EmailBackend"
```

in your settings.py.

MANDRILL_API_KEY

Required. Your Mandrill API key:

```
ANYMAIL = {
    ...
    "MANDRILL_API_KEY": "<your API key>",
}
```

Anymail will also look for `MANDRILL_API_KEY` at the root of the settings file if neither `ANYMAIL["MANDRILL_API_KEY"]` nor `ANYMAIL_MANDRILL_API_KEY` is set.

MANDRILL_WEBHOOK_KEY

Required if using Anymail's webhooks. The "webhook authentication key" issued by Mandrill. More info in Mandrill's KB.

MANDRILL_WEBHOOK_URL

Required only if using Anymail's webhooks *and* the hostname your Django server sees is different from the public webhook URL you provided Mandrill. (E.g., if you have a proxy in front of your Django server that forwards "https://yoursite.example.com" to "http://localhost:8000/").

If you are seeing `AnymailWebhookValidationFailure` errors from your webhooks, set this to the exact webhook URL you entered in Mandrill's settings.

MANDRILL_API_URL

The base url for calling the Mandrill API. The default is `MANDRILL_API_URL = "https://mandrillapp.com/api/1.0"`, which is the secure, production version of Mandrill's 1.0 API.

(It's unlikely you would need to change this.)

esp_extra support

To use Mandrill features not directly supported by Anymail, you can set a message's *`esp_extra`* to a `dict` of parameters to merge into Mandrill's messages/send API call. Note that a few parameters go at the top level, but Mandrill expects most options within a `'message'` sub-dict—be sure to check their API docs:

```
message.esp_extra = {
    # Mandrill expects 'ip_pool' at top level...
    'ip_pool': 'Bulk Pool',
    # ... but 'subaccount' must be within a 'message' dict:
    'message': {
        'subaccount': 'Marketing Dept.'
    }
}
```

Anymail has special handling that lets you specify Mandrill's `'recipient_metadata'` as a simple, pythonic `dict` (similar in form to Anymail's *merge_data*), rather than Mandrill's more complex list of rcpt/values dicts. You can use whichever style you prefer (but either way, recipient_metadata must be in `esp_extra['message']`).

Similary, Anymail allows Mandrill's `'template_content'` in esp_extra (top level) either as a pythonic `dict` (similar to Anymail's *merge_global_data*) or as Mandrill's more complex list of name/content dicts.

Limitations and quirks

Envelope sender uses only domain Anymail's *envelope_sender* is used to populate Mandrill's `'return_path_domain'`—but only the domain portion. (Mandrill always generates its own encoded mailbox for the envelope sender.)

Batch sending/merge and ESP templates

Mandrill offers both *ESP stored templates* and *batch sending* with per-recipient merge data.

You can use a Mandrill stored template by setting a message's *template_id* to the template's name. Alternatively, you can refer to merge fields directly in an EmailMessage's subject and body—the message itself is used as an on-the-fly template.

In either case, supply the merge data values with Anymail's normalized *merge_data* and *merge_global_data* message attributes.

```
# This example defines the template inline, using Mandrill's
# default MailChimp merge *|field|* syntax.
# You could use a stored template, instead, with:
#   message.template_id = "template name"
message = EmailMessage(
    ...
    subject="Your order *|order_no|* has shipped",
    body="""Hi *|name|*,
            We shipped your order *|order_no|*
            on *|ship_date|*.""",
    to=["alice@example.com", "Bob <bob@example.com>"]
)
# (you'd probably also set a similar html body with merge fields)
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15",
}
```

When you supply per-recipient *merge_data*, Anymail automatically forces Mandrill's `preserve_recipients` option to false, so that each person in the message's "to" list sees only their own email address.

To use the subject or from address defined with a Mandrill template, set the message's `subject` or `from_email` attribute to `None`.

See the Mandrill's template docs for more information.

Status tracking and inbound webhooks

If you are using Anymail's normalized *status tracking* and/or *inbound* handling, setting up Anymail's webhook URL requires deploying your Django project twice:

1. First, follow the instructions to *configure Anymail's webhooks*. You *must deploy* before adding the webhook URL to Mandrill, because Mandrill will attempt to verify the URL against your production server.

   Once you've deployed, then set Anymail's webhook URL in Mandrill, following their instructions for tracking event webhooks (be sure to check the boxes for the events you want to receive) and/or inbound route webhooks. In either case, the webhook url is:

   `https://random:random@yoursite.example.com/anymail/mandrill/`

   - *random:random* is an *ANYMAIL_WEBHOOK_SECRET* shared secret
   - *yoursite.example.com* is your Django site
   - (Note: Unlike Anymail's other supported ESPs, the Mandrill webhook uses this single url for both tracking and inbound events.)

2. Mandrill will provide you a "webhook authentication key" once it verifies the URL is working. Add this to your Django project's Anymail settings under *MANDRILL_WEBHOOK_KEY*. (You may also need to set *MANDRILL_WEBHOOK_URL* depending on your server config.) Then deploy your project again.

Mandrill implements webhook signing on the entire event payload, and Anymail verifies this signature. Until the correct webhook key is set, Anymail will raise an exception for any webhook calls from Mandrill (other than the initial validation request).

Mandrill's webhook signature also covers the exact posting URL. Anymail can usually figure out the correct (public) URL where Mandrill called your webhook. But if you're getting an `AnymailWebhookValidationFailure` with a different URL than you provided Mandrill, you may need to examine your Django `SECURE_PROXY_SSL_HEADER`, `USE_X_FORWARDED_HOST`, and/or `USE_X_FORWARDED_PORT` settings. If all else fails, you can set Anymail's *MANDRILL_WEBHOOK_URL* to the same public webhook URL you gave Mandrill.

Mandrill will report these Anymail *event_type*s: sent, rejected, deferred, bounced, opened, clicked, complained, unsubscribed, inbound. Mandrill does not support delivered events. Mandrill "whitelist" and "blacklist" change events will show up as Anymail's unknown event_type.

The event's *esp_event* field will be a `dict` of Mandrill event fields, for a single event. (Although Mandrill calls webhooks with batches of events, Anymail will invoke your signal receiver separately for each event in the batch.)

Changed in version 1.3: Earlier Anymail releases used `.../anymail/mandrill/tracking/` as the tracking webhook url. With the addition of inbound handling, Anymail has dropped "tracking" from the recommended url for new installations. But the older url is still supported. Existing installations can continue to use it—and can even install it on a Mandrill *inbound* route to avoid issuing a new webhook key.

Migrating from Djrill

Anymail has its origins as a fork of the Djrill package, which supported only Mandrill. If you are migrating from Djrill to Anymail – e.g., because you are thinking of switching ESPs – you'll need to make a few changes to your code.

Changes to settings

MANDRILL_API_KEY Will still work, but consider moving it into the *ANYMAIL* settings dict, or changing it to *ANYMAIL_MANDRILL_API_KEY*.

MANDRILL_SETTINGS Use *ANYMAIL_SEND_DEFAULTS* and/or ANYMAIL_MANDRILL_SEND_DEFAULTS (see *Global send defaults*).

There is one slight behavioral difference between *ANYMAIL_SEND_DEFAULTS* and Djrill's MANDRILL_SETTINGS: in Djrill, setting `tags` or `merge_vars` on a message would completely override any global settings defaults. In Anymail, those message attributes are merged with the values from *ANYMAIL_SEND_DEFAULTS*.

MANDRILL_SUBACCOUNT Set *esp_extra* globally in *ANYMAIL_SEND_DEFAULTS*:

```
ANYMAIL = {
    ...
    "MANDRILL_SEND_DEFAULTS": {
        "esp_extra": {
            "message": {
                "subaccount": "<your subaccount>"
            }
        }
    }
}
```

MANDRILL_IGNORE_RECIPIENT_STATUS Renamed to *ANYMAIL_IGNORE_RECIPIENT_STATUS* (or just IGNORE_RECIPIENT_STATUS in the *ANYMAIL* settings dict).

DJRILL_WEBHOOK_SECRET and DJRILL_WEBHOOK_SECRET_NAME Replaced with HTTP basic auth. See *Securing webhooks*.

DJRILL_WEBHOOK_SIGNATURE_KEY Use *ANYMAIL_MANDRILL_WEBHOOK_KEY* instead.

DJRILL_WEBHOOK_URL Often no longer required: Anymail can normally use Django's `HttpRequest.build_absolute_uri` to figure out the complete webhook url that Mandrill called.

If you are experiencing webhook authorization errors, the best solution is to adjust your Django `SECURE_PROXY_SSL_HEADER`, `USE_X_FORWARDED_HOST`, and/or `USE_X_FORWARDED_PORT` settings to work with your proxy server. If that's not possible, you can set *ANYMAIL_MANDRILL_WEBHOOK_URL* to explicitly declare the webhook url.

Changes to EmailMessage attributes

message.send_at If you are using an aware datetime for *send_at*, it will keep working unchanged with Anymail.

If you are using a date (without a time), or a naive datetime, be aware that these now default to Django's current_timezone, rather than UTC as in Djrill.

(As with Djrill, it's best to use an aware datetime that says exactly when you want the message sent.)

message.mandrill_response Anymail normalizes ESP responses, so you don't have to be familiar with the format of Mandrill's JSON. See *anymail_status*.

The *raw* ESP response is attached to a sent message as `anymail_status.esp_response`, so the direct replacement for message.mandrill_response is:

```
mandrill_response = message.anymail_status.esp_response.json()
```

message.template_name Anymail renames this to *template_id*.

message.merge_vars and message.global_merge_vars Anymail renames these to *merge_data* and *merge_global_data*, respectively.

message.use_template_from and message.use_template_subject With Anymail, set `message.from_email = None` or `message.subject = None` to use the values from the stored template.

message.return_path_domain With Anymail, set *envelope_sender* instead. You'll need to pass a valid email address (not just a domain), but Anymail will use only the domain, and will ignore anything before the @.

Changed in version 2.0.

Other Mandrill-specific attributes Djrill allowed nearly all Mandrill API parameters to be set as attributes directly on an EmailMessage. With Anymail, you should instead set these in the message's *esp_extra* dict as described above.

Although the Djrill style attributes are still supported (for now), Anymail will issue a `DeprecationWarning` if you try to use them. These warnings are visible during tests (with Django's default test runner), and will explain how to update your code.

You can also use the following git grep expression to find potential problems:

```
git grep -w \
  -e 'async' -e 'auto_html' -e 'auto_text' -e 'from_name' -e 'global_
↪merge_vars' \
  -e 'google_analytics_campaign' -e 'google_analytics_domains' -e
↪'important' \
  -e 'inline_css' -e 'ip_pool' -e 'merge_language' -e 'merge_vars' \
  -e 'preserve_recipients' -e 'recipient_metadata' -e 'return_path_domain
↪' \
  -e 'signing_domain' -e 'subaccount' -e 'template_content' -e 'template_
↪name' \
  -e 'tracking_domain' -e 'url_strip_qs' -e 'use_template_from' -e 'use_
↪template_subject' \
  -e 'view_content_link'
```

Inline images Djrill (incorrectly) used the presence of a *Content-ID* header to decide whether to treat an image as inline. Anymail looks for *Content-Disposition: inline*.

If you were constructing MIMEImage inline image attachments for your Djrill messages, in addition to setting the Content-ID, you should also add:

```
image.add_header('Content-Disposition', 'inline')
```

Or better yet, use Anymail's new *Inline images* helper functions to attach your inline images.

Changes to webhooks

Anymail uses HTTP basic auth as a shared secret for validating webhook calls, rather than Djrill's "secret" query parameter. See *Securing webhooks*. (A slight advantage of basic auth over query parameters is that most logging and analytics systems are aware of the need to keep auth secret.)

Anymail replaces `djrill.signals.webhook_event` with `anymail.signals.tracking` for delivery tracking events, and `anymail.signals.inbound` for inbound events. Anymail parses and normalizes the event data passed to the signal receiver: see *Tracking sent mail status* and *Receiving mail*.

The equivalent of Djrill's `data` parameter is available to your signal receiver as *`event.esp_event`*, and for most events, the equivalent of Djrill's `event_type` parameter is `event.esp_event['event']`. But consider working with Anymail's normalized *`AnymailTrackingEvent`* and *`AnymailInboundEvent`* instead for easy portability to other ESPs.

1.5.5 Postmark

Anymail integrates with the Postmark transactional email service, using their HTTP email API.

Settings

EMAIL_BACKEND

To use Anymail's Postmark backend, set:

```
EMAIL_BACKEND = "anymail.backends.postmark.EmailBackend"
```

in your settings.py.

POSTMARK_SERVER_TOKEN

Required. A Postmark server token.

```
ANYMAIL = {
    ...
    "POSTMARK_SERVER_TOKEN": "<your server token>",
}
```

Anymail will also look for `POSTMARK_SERVER_TOKEN` at the root of the settings file if neither `ANYMAIL["POSTMARK_SERVER_TOKEN"]` nor `ANYMAIL_POSTMARK_SERVER_TOKEN` is set.

You can override the server token for an individual message in its *esp_extra*.

POSTMARK_API_URL

The base url for calling the Postmark API.

The default is `POSTMARK_API_URL = "https://api.postmarkapp.com/"` (It's unlikely you would need to change this.)

esp_extra support

To use Postmark features not directly supported by Anymail, you can set a message's *`esp_extra`* to a `dict` that will be merged into the json sent to Postmark's email API.

Example:

```
message.esp_extra = {
    'HypotheticalFuturePostmarkParam': '2022',  # merged into send params
    'server_token': '<API server token for just this message>',
}
```

(You can also set `"esp_extra"` in Anymail's *global send defaults* to apply it to all messages.)

Limitations and quirks

Postmark does not support a few tracking and reporting additions offered by other ESPs.

Anymail normally raises an *AnymailUnsupportedFeature* error when you try to send a message using features that Postmark doesn't support You can tell Anymail to suppress these errors and send the messages anyway – see *Unsupported features*.

Single tag Postmark allows a maximum of one tag per message. If your message has two or more *tags*, you'll get an *AnymailUnsupportedFeature* error—or if you've enabled *ANYMAIL_IGNORE_UNSUPPORTED_FEATURES*, Anymail will use only the first tag.

No delayed sending Postmark does not support *send_at*.

Click-tracking Postmark supports several link-tracking options. Anymail treats *track_clicks* as Postmark's "HtmlAndText" option when True.

If you would prefer Postmark's "HtmlOnly" or "TextOnly" link-tracking, you could either set that as a Postmark server-level default (and use `message.track_clicks = False` to disable tracking for specific messages), or use something like `message.esp_extra = {'TrackLinks': "HtmlOnly"}` to specify a particular option.

No envelope sender overrides Postmark does not support overriding *envelope_sender* on individual messages. (You can configure custom return paths for each sending domain in the Postmark control panel.)

Batch sending/merge and ESP templates

Postmark offers both *ESP stored templates* and *batch sending* with per-recipient merge data.

Changed in version 4.2: Added Postmark *merge_data* and batch sending support. (Earlier Anymail releases only supported *merge_global_data* with Postmark.)

To use a Postmark template, set the message's *template_id* to either the numeric Postmark "TemplateID" or its string "TemplateAlias" (which is *not* the template's name). You can find a template's numeric id near the top right in Postmark's template editor, and set the alias near the top right above the name.

Changed in version 5.0: Earlier Anymail releases only allowed numeric template IDs.

Supply the Postmark "TemplateModel" variables using Anymail's normalized *merge_data* and *merge_global_data* message attributes:

```
message = EmailMessage(
    # (subject and body come from the template, so don't include those)
    to=["alice@example.com", "Bob <bob@example.com>"]
)
message.template_id = 80801  # Postmark template id or alias
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15",
}
```

Postmark does not allow overriding the message's subject or body with a template. (You can customize the subject by including variables in the template's subject.)

When you supply per-recipient *merge_data*, Anymail automatically switches to Postmark's batch send API, so that each "to" recipient sees only their own email address. (Any cc's or bcc's will be duplicated for *every* to-recipient.)

If you want to use batch sending with a regular message (without a template), set merge data to an empty dict: `message.merge_data = {}`.

See this Postmark blog post on templates for more information.

Status tracking webhooks

If you are using Anymail's normalized *status tracking*, set up a webhook in your Postmark account settings, under Servers > *your server name* > Settings > Webhooks. The webhook URL is:

`https://random:random@yoursite.example.com/anymail/postmark/tracking/`

- *random:random* is an `ANYMAIL_WEBHOOK_SECRET` shared secret
- *yoursite.example.com* is your Django site

Choose all the event types you want to receive. Anymail doesn't care about the "include messsage content" and "post only on first open" options; whether to use them is your choice.

If you use multiple Postmark servers, you'll need to repeat entering the webhook settings for each of them.

Postmark will report these Anymail `event_type`s: rejected, failed, bounced, deferred, delivered, autoresponded, opened, clicked, complained, unsubscribed, subscribed. (Postmark does not support sent–what it calls "processed"–events through webhooks.)

The event's `esp_event` field will be a `dict` of Postmark delivery, bounce, spam-complaint, open-tracking, or click data.

Inbound webhook

If you want to receive email from Postmark through Anymail's normalized *inbound* handling, follow Postmark's Inbound Processing guide to configure an inbound server pointing to Anymail's inbound webhook.

The InboundHookUrl setting will be:

`https://random:random@yoursite.example.com/anymail/postmark/inbound/`

- *random:random* is an `ANYMAIL_WEBHOOK_SECRET` shared secret
- *yoursite.example.com* is your Django site

Anymail handles the "parse an email" part of Postmark's instructions for you, but you'll likely want to work through the other sections to set up a custom inbound domain, and perhaps configure inbound spam blocking.

1.5.6 SendGrid

Anymail integrates with the SendGrid email service, using their Web API v3.

Important: Troubleshooting: If your SendGrid messages aren't being delivered as expected, be sure to look for "drop" events in your SendGrid activity feed.

SendGrid detects certain types of errors only *after* the send API call appears to succeed, and reports these errors as drop events.

Settings

EMAIL_BACKEND

To use Anymail's SendGrid backend, set:

```
EMAIL_BACKEND = "anymail.backends.sendgrid.EmailBackend"
```

in your settings.py.

SENDGRID_API_KEY

A SendGrid API key with "Mail Send" permission. (Manage API keys in your SendGrid API key settings.) Required.

```
ANYMAIL = {
    ...
    "SENDGRID_API_KEY": "<your API key>",
}
```

Anymail will also look for `SENDGRID_API_KEY` at the root of the settings file if neither `ANYMAIL["SENDGRID_API_KEY"]` nor `ANYMAIL_SENDGRID_API_KEY` is set.

SENDGRID_GENERATE_MESSAGE_ID

Whether Anymail should generate a UUID for each message sent through SendGrid, to facilitate status tracking. The UUID is attached to the message as a SendGrid custom arg named "anymail_id" and made available as *`anymail_status.message_id`* on the sent message.

Default `True`. You can set to `False` to disable this behavior, in which case sent messages will have a `message_id` of `None`. See *Message-ID quirks* below.

SENDGRID_MERGE_FIELD_FORMAT

If you use *merge data* with SendGrid's legacy transactional templates, set this to a `str.format()` formatting string that indicates how merge fields are delimited in your legacy templates. For example, if your templates use the `-field-` hyphen delimiters suggested in some SendGrid docs, you would set:

```
ANYMAIL = {
    ...
    "SENDGRID_MERGE_FIELD_FORMAT": "-{}-",
}
```

The placeholder `{}` will become the merge field name. If you need to include a literal brace character, double it up. (For example, Handlebars-style `{{field}}` delimiters would take the format string `"{{{{{}}}}}"`.)

The default `None` requires you include the delimiters directly in your *`merge_data`* keys. You can also override this setting for individual messages. See the notes on SendGrid *templates and merge* below.

This setting is not used (or necessary) with SendGrid's newer dynamic transactional templates, which always use Handlebars syntax.

SENDGRID_API_URL

The base url for calling the SendGrid API.

The default is `SENDGRID_API_URL = "https://api.sendgrid.com/v3/"` (It's unlikely you would need to change this.)

esp_extra support

To use SendGrid features not directly supported by Anymail, you can set a message's *esp_extra* to a `dict` of parameters for SendGrid's v3 Mail Send API. Your `esp_extra` dict will be deeply merged into the parameters Anymail has constructed for the send, with `esp_extra` having precedence in conflicts.

Anymail has special handling for `esp_extra["personalizations"]`. If that value is a `dict`, Anymail will merge that personalizations dict into the personalizations for each message recipient. (If you pass a `list`, that will override the personalizations Anymail normally constructs from the message, and you will need to specify each recipient in the personalizations list yourself.)

Example:

```
message.open_tracking = True
message.esp_extra = {
    "asm": {  # SendGrid subscription management
        "group_id": 1,
        "groups_to_display": [1, 2, 3],
    },
    "tracking_settings": {
        "open_tracking": {
            # Anymail will automatically set `"enable": True` here,
            # based on message.open_tracking.
            "substitution_tag": "%%OPEN_TRACKING_PIXEL%%",
        },
    },
    # Because "personalizations" is a dict, Anymail will merge "future_
↪feature"
    # into the SendGrid personalizations array for each message recipient
    "personalizations": {
        "future_feature": {"future": "data"},
    },
}
```

(You can also set `"esp_extra"` in Anymail's *global send defaults* to apply it to all messages.)

Limitations and quirks

Message-ID SendGrid does not return any sort of unique id from its send API call. Knowing a sent message's ID can be important for later queries about the message's status.

To work around this, Anymail generates a UUID for each outgoing message, provides it to SendGrid as a custom arg named "anymail_id" and makes it available as the message's *anymail_status.message_id* attribute after sending. The same UUID will be passed to Anymail's *tracking webhooks* as *event.message_id*.

To disable attaching tracking UUIDs to sent messages, set *SENDGRID_GENERATE_MESSAGE_ID* to False in your Anymail settings.

Changed in version 6.0: In batch sends, Anymail generates a distinct anymail_id for *each* "to" recipient. (Previously, a single id was used for all batch recipients.) Check *anymail_status.recipients[to_email].message_id* for individual batch-send tracking ids.

Changed in version 3.0: Previously, Anymail generated a custom *Message-ID* header for each sent message. But SendGrid's "smtp-id" event field does not reliably reflect this header, which complicates status tracking. (For compatibility with messages sent in earlier versions, Anymail's webhook `message_id` will fall back to "smtp-id" when "anymail_id" isn't present.)

Single Reply-To SendGrid's v3 API only supports a single Reply-To address.

If your message has multiple reply addresses, you'll get an *AnymailUnsupportedFeature* error—or if you've enabled *ANYMAIL_IGNORE_UNSUPPORTED_FEATURES*, Anymail will use only the first one.

Invalid Addresses SendGrid will accept *and send* just about anything as a message's `from_email`. (And email protocols are actually OK with that.)

(Tested March, 2016)

No envelope sender overrides SendGrid does not support overriding *envelope_sender* on individual messages.

Batch sending/merge and ESP templates

SendGrid offers both *ESP stored templates* and *batch sending* with per-recipient merge data.

SendGrid has two types of stored templates for transactional email:

- Dynamic transactional templates, which were introduced in July, 2018, use Handlebars template syntax and allow complex logic to be coded in the template itself.
- Legacy transactional templates, which allow only simple key-value substitution and don't specify a particular template syntax.

[Legacy templates were originally just called "transactional templates," and many older references still use this terminology. But confusingly, SendGrid's dashboard and some recent articles now use "transactional templates" to mean the newer, dynamic templates.]

Changed in version 4.1: Added support for SendGrid dynamic transactional templates. (Earlier Anymail releases work only with SendGrid's legacy transactional templates.)

You can use either type of SendGrid stored template by setting a message's *template_id* to the template's unique id (*not* its name). Supply the merge data values with Anymail's normalized *merge_data* and *merge_global_data* message attributes.

```
message = EmailMessage(
    ...
    # omit subject and body (or set to None) to use template content
    to=["alice@example.com", "Bob <bob@example.com>"]
)
message.template_id = "d-5a963add2ec84305813ff860db277d7a"  # SendGrid␣
↪dynamic id
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15",
}
```

When you supply per-recipient *merge_data*, Anymail automatically changes how it communicates the "to" list to SendGrid, so that each recipient sees only their own email address. (Anymail creates a separate "personalization" for each recipient in the "to" list; any cc's or bcc's will be duplicated for *every* to-recipient.)

See the SendGrid's transactional template overview for more information.

Legacy transactional templates

With *legacy* transactional templates (only), SendGrid doesn't have a pre-defined merge field syntax, so you must tell Anymail how substitution fields are delimited in your templates. There are three ways you can do this:

- Set `'merge_field_format'` in the message's *esp_extra* to a python `str.format()` string, as shown in the example below. (This applies only to that particular EmailMessage.)
- *Or* set *SENDGRID_MERGE_FIELD_FORMAT* in your Anymail settings. This is usually the best approach, and will apply to all legacy template messages sent through SendGrid. (You can still use esp_extra to override for individual messages.)
- *Or* include the field delimiters directly in *all* your *merge_data* and *merge_global_data* keys. E.g.: `{'-name-': "Alice", '-order_no-': "12345"}`. (This can be error-prone, and makes it difficult to transition to other ESPs or to SendGrid's dynamic templates.)

```
# ...
message.template_id = "5997fcf6-2b9f-484d-acd5-7e9a99f0dc1f"  # SendGrid␣
↪legacy id
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.esp_extra = {
    # Tell Anymail this SendGrid legacy template uses "-field-" for merge␣
↪fields.
    # (You could instead set SENDGRID_MERGE_FIELD_FORMAT in your ANYMAIL␣
↪settings.)
    'merge_field_format': "-{}-"
}
```

SendGrid legacy templates allow you to mix your EmailMessage's `subject` and `body` with the template subject and body (by using `<%subject%>` and `<%body%>` in your SendGrid template definition where you want the message-specific versions to appear). If you don't want to supply any additional subject or body content from your Django app, set those EmailMessage attributes to empty strings or `None`.

On-the-fly templates

Rather than define a stored ESP template, you can refer to merge fields directly in an EmailMessage's subject and body, and SendGrid will treat this as an on-the-fly, legacy-style template definition. (The on-the-fly template can't contain any dynamic template logic, and like any legacy template you must specify the merge field format in either Anymail settings or esp_extra as described above.)

```
# on-the-fly template using merge fields in subject and body:
message = EmailMessage(
    subject="Your order {{order_no}} has shipped",
    body="Dear {{name}}:\nWe've shipped order {{order_no}}.",
    to=["alice@example.com", "Bob <bob@example.com>"]
)
# note: no template_id specified
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.esp_extra = {
    # here's how to get Handlebars-style {{merge}} fields with Python's str.
↪format:
```

(continues on next page)

(continued from previous page)

```
    'merge_field_format': "{{{{{}}}}}"  # "{{ {{ {} }} }}" without the spaces
}
```

Status tracking webhooks

If you are using Anymail's normalized *status tracking*, enter the url in your SendGrid mail settings, under "Event Notification":

`https://random:random@yoursite.example.com/anymail/sendgrid/tracking/`

- *random:random* is an `ANYMAIL_WEBHOOK_SECRET` shared secret
- *yoursite.example.com* is your Django site

Be sure to check the boxes in the SendGrid settings for the event types you want to receive.

SendGrid will report these Anymail `event_type`s: queued, rejected, bounced, deferred, delivered, opened, clicked, complained, unsubscribed, subscribed.

The event's `esp_event` field will be a `dict` of Sendgrid event fields, for a single event. (Although SendGrid calls webhooks with batches of events, Anymail will invoke your signal receiver separately for each event in the batch.)

Inbound webhook

If you want to receive email from SendGrid through Anymail's normalized *inbound* handling, follow SendGrid's Inbound Parse Webhook guide to set up Anymail's inbound webhook.

The Destination URL setting will be:

`https://random:random@yoursite.example.com/anymail/sendgrid/inbound/`

- *random:random* is an `ANYMAIL_WEBHOOK_SECRET` shared secret
- *yoursite.example.com* is your Django site

Be sure the URL has a trailing slash. (SendGrid's inbound processing won't follow Django's `APPEND_SLASH` redirect.)

If you want to use Anymail's normalized `spam_detected` and `spam_score` attributes, be sure to enable the "Check incoming emails for spam" checkbox.

You have a choice for SendGrid's "POST the raw, full MIME message" checkbox. Anymail will handle either option (and you can change it at any time). Enabling raw MIME will give the most accurate representation of *any* received email (including complex forms like multi-message mailing list digests). But disabling it *may* use less memory while processing messages with many large attachments.

1.5.7 SendinBlue

Anymail integrates with the SendinBlue email service, using their API v3. SendinBlue's transactional API does not support some basic email features, such as inline images. Be sure to review the *limitations* below.

Important: Troubleshooting: If your SendinBlue messages aren't being delivered as expected, be sure to look for events in your SendinBlue logs.

SendinBlue detects certain types of errors only *after* the send API call reports the message as "queued." These errors appear in the logging dashboard.

Settings

EMAIL_BACKEND

To use Anymail's SendinBlue backend, set:

```
EMAIL_BACKEND = "anymail.backends.sendinblue.EmailBackend"
```

in your settings.py.

SENDINBLUE_API_KEY

The API key can be retrieved from your SendinBlue SMTP & API settings. Make sure the version column indicates "v3." (v2 keys don't work with Anymail. If you don't see a v3 key listed, use "Create a New API Key".) Required.

```
ANYMAIL = {
    ...
    "SENDINBLUE_API_KEY": "<your v3 API key>",
}
```

Anymail will also look for `SENDINBLUE_API_KEY` at the root of the settings file if neither `ANYMAIL["SENDINBLUE_API_KEY"]` nor `ANYMAIL_SENDINBLUE_API_KEY` is set.

SENDINBLUE_API_URL

The base url for calling the SendinBlue API.

The default is `SENDINBLUE_API_URL = "https://api.sendinblue.com/v3/"` (It's unlikely you would need to change this.)

esp_extra support

To use SendinBlue features not directly supported by Anymail, you can set a message's *`esp_extra`* to a `dict` that will be merged into the json sent to SendinBlue's smtp/email API.

Example:

```
message.esp_extra = {
    'hypotheticalFutureSendinBlueParam': '2022',  # merged into send params
}
```

(You can also set `"esp_extra"` in Anymail's *global send defaults* to apply it to all messages.)

Limitations and quirks

SendinBlue's v3 API has several limitations. In most cases below, Anymail will raise an *`AnymailUnsupportedFeature`* error if you try to send a message using missing features. You can override this by enabling the *`ANYMAIL_IGNORE_UNSUPPORTED_FEATURES`* setting, and Anymail will try to limit the API request to features SendinBlue can handle.

HTML body required SendinBlue's API returns an error if you attempt to send a message with only a plain-text body. Be sure to *include HTML* content for your messages if you are not using a template.

(SendinBlue *does* allow HTML without a plain-text body. This is generally not recommended, though, as some email systems treat HTML-only content as a spam signal.)

Inline images SendinBlue's v3 API doesn't support inline images, at all. (Confirmed with SendinBlue support Feb 2018.)

If you are ignoring unsupported features, Anymail will try to send inline images as ordinary image attachments.

Attachment names must be filenames with recognized extensions SendinBlue determines attachment content type by assuming the attachment's name is a filename, and examining that filename's extension (e.g., ".jpg").

Trying to send an attachment without a name, or where the name does not end in a supported filename extension, will result in a SendinBlue API error. Anymail has no way to communicate an attachment's desired content-type to the SendinBlue API if the name is not set correctly.

No attachments with templates If you are sending using a SendinBlue template, their API doesn't support ordinary file attachments. Attempting to send an attachment with a template will result in the SendinBlue API error message, "Please don't pass attachment content & templateId in same request, instead use attachment url only." See the *templates* section below.

Single Reply-To SendinBlue's v3 API only supports a single Reply-To address.

If you are ignoring unsupported features and have multiple reply addresses, Anymail will use only the first one.

Metadata Anymail passes *metadata* to SendinBlue as a JSON-encoded string using their *X-Mailin-custom* email header. The metadata is available in tracking webhooks.

No delayed sending SendinBlue does not support *send_at*.

No click-tracking or open-tracking options SendinBlue does not provide a way to control open or click tracking for individual messages. Anymail's *track_clicks* and *track_opens* settings are unsupported.

No envelope sender overrides SendinBlue does not support overriding *envelope_sender* on individual messages.

Batch sending/merge and ESP templates

SendinBlue supports *ESP stored templates* populated with global merge data for all recipients, but does not offer *batch sending* with per-recipient merge data. Anymail's *merge_data* and *merge_metadata* message attributes are not supported with the SendinBlue backend, but you can use Anymail's *merge_global_data* with SendinBlue templates.

SendinBlue supports two different template styles: a new template language that uses Django template syntax (with `{{ param.NAME }}` style substitutions), and an "old" template language that used percent-delimited `%NAME%` style substitutions. Anymail v7.0 and later require new style templates.

Changed in version 7.0: Anymail switched to a SendinBlue API that supports the new template language and removes several limitations from the earlier template send API. But the new API does not support attachments, and can behave oddly if used with old style templates.

Caution: Anymail v7.0 and later work *only* with Sendinblue's *new* template language. You should follow SendinBlue's instructions to convert each old template to the new language.

Although unconverted old templates may appear to work with Anymail v7.0, some features may not work properly. In particular, `reply_to` overrides and recipient display names are silently ignored when *old* style templates are sent with the *new* API used in Anymail v7.0.

To use a SendinBlue template, set the message's *template_id* to the numeric SendinBlue template ID, and supply substitution attributes using the message's *merge_global_data*:

```
message = EmailMessage(
    to=["alice@example.com"]  # single recipient...
    # ...multiple to emails would all get the same message
    # (and would all see each other's emails in the "to" header)
)
message.template_id = 3   # use this SendinBlue template
message.from_email = None  # to use the template's default sender
message.merge_global_data = {
    'name': "Alice",
    'order_no': "12345",
    'ship_date': "May 15",
}
```

Within your SendinBlue template body and subject, you can refer to merge variables using Django template syntax, like `{{ params.order_no }}` or `{{ params.ship_date }}` for the example above.

The message's `from_email` (which defaults to your `DEFAULT_FROM_EMAIL` setting) will override the template's default sender. If you want to use the template's sender, be sure to set `from_email` to `None` *after* creating the message, as shown in the example above.

You can also override the template's subject and reply-to address (but not body) using standard `EmailMessage` attributes.

SendinBlue's template feature does not currently support providing attachment content directly with the message—you'll get a SendinBlue API error if you try. If you must send file attachments with SendinBlue templates, you can either upload them into SendinBlue's template designer, or arrange to have the attachment content hosted on a public URL and use Anymail's *esp_extra* to pass the URL to the SendinBlue API (see the Anymail SendinBlue integration tests for an example of this). A better—and portable—option may be to avoid SendinBlue templates and instead render the email in your Django code, allowing you to add any file attachments you want. See *Using Django templates for email* for details.

(Note that SendinBlue doesn't support *inline* image attachments at all, whether you're using a template or not.)

Status tracking webhooks

If you are using Anymail's normalized *status tracking*, add the url at SendinBlue's site under Transactional > Settings > Webhook.

The "URL to call" is:

`https://random:random@yoursite.example.com/anymail/sendinblue/tracking/`

- *random:random* is an *ANYMAIL_WEBHOOK_SECRET* shared secret
- *yoursite.example.com* is your Django site

Be sure to select the checkboxes for all the event types you want to receive. (Also make sure you are in the "Transactional" section of their site; SendinBlue has a separate set of "Campaign" webhooks, which don't apply to messages sent through Anymail.)

If you are interested in tracking opens, note that SendinBlue has both a "First opening" and an "Opened" event type, and will generate both the first time a message is opened. Anymail normalizes both of these events to "opened." To avoid double counting, you should only enable one of the two.

SendinBlue will report these Anymail *event_type*s: queued, rejected, bounced, deferred, delivered, opened (see note above), clicked, complained, unsubscribed, subscribed (though this should never occur for transactional email).

For events that occur in rapid succession, SendinBlue frequently delivers them out of order. For example, it's not uncommon to receive a "delivered" event before the corresponding "queued."

The event's *esp_event* field will be a dict of raw webhook data received from SendinBlue.

Inbound webhook

SendinBlue does not support inbound email handling.

1.5.8 SparkPost

Anymail integrates with the SparkPost email service, using their Python sparkpost API client package.

Installation

You must ensure the sparkpost package is installed to use Anymail's SparkPost backend. Either include the "sparkpost" option when you install Anymail:

```
$ pip install django-anymail[sparkpost]
```

or separately run `pip install sparkpost`.

Settings

EMAIL_BACKEND

To use Anymail's SparkPost backend, set:

```
EMAIL_BACKEND = "anymail.backends.sparkpost.EmailBackend"
```

in your settings.py.

SPARKPOST_API_KEY

A SparkPost API key with at least the "Transmissions: Read/Write" permission. (Manage API keys in your SparkPost account API keys.)

This setting is optional; if not provided, the SparkPost API client will attempt to read your API key from the `SPARKPOST_API_KEY` environment variable.

```
ANYMAIL = {
    ...
    "SPARKPOST_API_KEY": "<your API key>",
}
```

Anymail will also look for `SPARKPOST_API_KEY` at the root of the settings file if neither `ANYMAIL["SPARKPOST_API_KEY"]` nor `ANYMAIL_SPARKPOST_API_KEY` is set.

SPARKPOST_API_URL

The SparkPost API Endpoint to use. This setting is optional; if not provided, Anymail will use the python-sparkpost client default endpoint (`"https://api.sparkpost.com/api/v1"`).

Set this to use a SparkPost EU account, or to work with any other API endpoint including SparkPost Enterprise API and SparkPost Labs.

```
ANYMAIL = {
    ...
    "SPARKPOST_API_URL": "https://api.eu.sparkpost.com/api/v1",  # use␣
↪SparkPost EU
}
```

You must specify the full, versioned API endpoint as shown above (not just the base_uri). This setting only affects Anymail's calls to SparkPost, and will not apply to other code using python-sparkpost.

esp_extra support

To use SparkPost features not directly supported by Anymail, you can set a message's *esp_extra* to a `dict` of parameters for python-sparkpost's transmissions.send method. Any keys in your `esp_extra` dict will override Anymail's normal values for that parameter.

Example:

```
message.esp_extra = {
    'transactional': True,  # treat as transactional for unsubscribe and␣
↪suppression
    'description': "Marketing test-run for new templates",
    'use_draft_template': True,
}
```

(You can also set `"esp_extra"` in Anymail's *global send defaults* to apply it to all messages.)

Limitations and quirks

Anymail's 'message_id' is SparkPost's 'transmission_id' The *message_id* Anymail sets on a message's *anymail_status* and in normalized webhook *AnymailTrackingEvent* data is actually what SparkPost calls "transmission_id".

Like Anymail's message_id for other ESPs, SparkPost's transmission_id (together with the recipient email address), uniquely identifies a particular message instance in tracking events.

(The transmission_id is the only unique identifier available when you send your message. SparkPost also has something called "message_id", but that doesn't get assigned until after the send API call has completed.)

If you are working exclusively with Anymail's normalized message status and webhook events, the distinction won't matter: you can consistently use Anymail's `message_id`. But if you are also working with raw webhook esp_event data or SparkPost's events API, be sure to think "transmission_id" wherever you're speaking to SparkPost.

Single tag Anymail uses SparkPost's "campaign_id" to implement message tagging. SparkPost only allows a single campaign_id per message. If your message has two or more *tags*, you'll get an *AnymailUnsupportedFeature* error—or if you've enabled *ANYMAIL_IGNORE_UNSUPPORTED_FEATURES*, Anymail will use only the first tag.

(SparkPost's "recipient tags" are not available for tagging *messages*. They're associated with individual *addresses* in stored recipient lists.)

Envelope sender may use domain only Anymail's *envelope_sender* is used to populate SparkPost's `'return_path'` parameter. Anymail supplies the full email address, but depending on your SparkPost configuration, SparkPost may use only the domain portion and substitute its own encoded mailbox before the @.

Batch sending/merge and ESP templates

SparkPost offers both *ESP stored templates* and *batch sending* with per-recipient merge data.

You can use a SparkPost stored template by setting a message's *template_id* to the template's unique id. (When using a stored template, SparkPost prohibits setting the EmailMessage's subject, text body, or html body.)

Alternatively, you can refer to merge fields directly in an EmailMessage's subject, body, and other fields—the message itself is used as an on-the-fly template.

In either case, supply the merge data values with Anymail's normalized *merge_data* and *merge_global_data* message attributes.

```
message = EmailMessage(
    ...
    to=["alice@example.com", "Bob <bob@example.com>"]
)
message.template_id = "11806290401558530"  # SparkPost id
message.merge_data = {
    'alice@example.com': {'name': "Alice", 'order_no': "12345"},
    'bob@example.com': {'name': "Bob", 'order_no': "54321"},
}
message.merge_global_data = {
    'ship_date': "May 15",
    # Can use SparkPost's special "dynamic" keys for nested substitutions␣
↪(see notes):
    'dynamic_html': {
        'status_html': "<a href='https://example.com/order/{{order_no}}'>
↪Status</a>",
    },
    'dynamic_plain': {
        'status_plain': "Status: https://example.com/order/{{order_no}}",
    },
}
```

See SparkPost's substitutions reference for more information on templates and batch send with SparkPost. If you need the special "dynamic" keys for nested substitutions, provide them in Anymail's *merge_global_data* as shown in the example above. And if you want `use_draft_template` behavior, specify that in *esp_extra*.

Status tracking webhooks

If you are using Anymail's normalized *status tracking*, set up the webhook in your SparkPost account settings under "Webhooks":

- Target URL: `https://yoursite.example.com/anymail/sparkpost/tracking/`
- Authentication: choose "Basic Auth." For username and password enter the two halves of the *random:random* shared secret you created for your *ANYMAIL_WEBHOOK_SECRET* Django setting. (Anymail doesn't support OAuth webhook auth.)
- Events: click "Select" and then *clear* the checkbox for "Relay Events" category (which is for inbound email). You can leave all the other categories of events checked, or disable any you aren't interested in tracking.

SparkPost will report these Anymail *event_type*s: queued, rejected, bounced, deferred, delivered, opened, clicked, complained, unsubscribed, subscribed.

The event's *esp_event* field will be a single, raw SparkPost event. (Although SparkPost calls webhooks with batches of events, Anymail will invoke your signal receiver separately for each event in the batch.) The esp_event is

the raw, wrapped json event structure as provided by SparkPost: `{'msys': {'<event_category>': {...<actual event data>...}}}`.

Inbound webhook

If you want to receive email from SparkPost through Anymail's normalized *inbound* handling, follow SparkPost's Enabling Inbound Email Relaying guide to set up Anymail's inbound webhook.

The target parameter for the Relay Webhook will be:

`https://random:random@yoursite.example.com/anymail/sparkpost/inbound/`

- *random:random* is an *ANYMAIL_WEBHOOK_SECRET* shared secret
- *yoursite.example.com* is your Django site

1.5.9 Anymail feature support

The table below summarizes the Anymail features supported for each ESP.

Email Service Provider	*Amazon SES*	*Mailgun*	*Mailjet*	*Mandrill*	*Post-mark*	*Send-Grid*	*Sendin-Blue*	*Spark-Post*
Anymail send options								
envelope_sender	Yes	Domain only	Yes	Domain only	No	No	No	Yes
metadata	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
merge_metadata	No	Yes	Yes	Yes	Yes	Yes	No	Yes
send_at	No	Yes	No	Yes	No	Yes	No	Yes
tags	Yes	Yes	Max 1 tag	Yes	Max 1 tag	Yes	Yes	Max 1 tag
track_clicks	No	Yes	Yes	Yes	Yes	Yes	No	Yes
track_opens	No	Yes	Yes	Yes	Yes	Yes	No	Yes
Batch sending/merge and ESP templates								
template_id	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
merge_data	Yes	Yes	Yes	Yes	No	Yes	No	Yes
merge_global_data	Yes	Yes (emulated)	Yes	Yes	Yes	Yes	Yes	Yes
Status and event tracking								
anymail_status	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
AnymailTrackingEvent from webhooks	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Inbound handling								
AnymailInboundEvent from webhooks	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes

Trying to choose an ESP? Please **don't** start with this table. It's far more important to consider things like an ESP's deliverability stats, latency, uptime, and support for developers. The *number* of extra features an ESP offers is almost meaningless. (And even specific features don't matter if you don't plan to use them.)

1.5.10 Other ESPs

Don't see your favorite ESP here? Anymail is designed to be extensible. You can suggest that Anymail add an ESP, or even contribute your own implementation to Anymail. See *Contributing*.

1.6 Tips, tricks, and advanced usage

Some suggestions and recipes for getting things done with Anymail:

1.6.1 Handling transient errors

Applications using Anymail need to be prepared to deal with connectivity issues and other transient errors from your ESP's API (as with any networked API).

Because Django doesn't have a built-in way to say "try this again in a few moments," Anymail doesn't have its own logic to retry network errors. The best way to handle transient ESP errors depends on your Django project:

- If you already use something like celery or Django channels for background task scheduling, that's usually the best choice for handling Anymail sends. Queue a task for every send, and wait to mark the task complete until the send succeeds (or repeatedly fails, according to whatever logic makes sense for your app).
- Another option is the Pinax django-mailer package, which queues and automatically retries failed sends for any Django EmailBackend, including Anymail. django-mailer maintains its send queue in your regular Django DB, which is a simple way to get started but may not scale well for very large volumes of outbound email.

In addition to handling connectivity issues, either of these approaches also has the advantage of moving email sending to a background thread. This is a best practice for sending email from Django, as it allows your web views to respond faster.

1.6.2 Mixing email backends

Since you are replacing Django's global `EMAIL_BACKEND`, by default Anymail will handle **all** outgoing mail, sending everything through your ESP.

You can use Django mail's optional `connection` argument to send some mail through your ESP and others through a different system.

This could be useful, for example, to deliver customer emails with the ESP, but send admin emails directly through an SMTP server:

```
from django.core.mail import send_mail, get_connection

# send_mail connection defaults to the settings EMAIL_BACKEND, which
# we've set to Anymail's Mailgun EmailBackend. This will be sent using Mailgun:
send_mail("Thanks", "We sent your order", "sales@example.com", ["customer@example.com
↪"])

# Get a connection to an SMTP backend, and send using that instead:
smtp_backend = get_connection('django.core.mail.backends.smtp.EmailBackend')
```

(continues on next page)

(continued from previous page)

```
send_mail("Uh-Oh", "Need your attention", "admin@example.com", ["alert@example.com"],
          connection=smtp_backend)

# You can even use multiple Anymail backends in the same app:
sendgrid_backend = get_connection('anymail.backends.sendgrid.EmailBackend')
send_mail("Password reset", "Here you go", "noreply@example.com", ["user@example.com
↪"],
          connection=sendgrid_backend)

# You can override settings.py settings with kwargs to get_connection.
# This example supplies credentials for a different Mailgun sub-acccount:
alt_mailgun_backend = get_connection('anymail.backends.mailgun.EmailBackend',
                                     api_key=MAILGUN_API_KEY_FOR_MARKETING)
send_mail("Here's that info", "you wanted", "info@marketing.example.com", [
↪"prospect@example.org"],
          connection=alt_mailgun_backend)
```

You can supply a different connection to Django's `send_mail()` and `send_mass_mail()` helpers, and in the constructor for an `EmailMessage` or `EmailMultiAlternatives`.

(See the `django.utils.log.AdminEmailHandler` docs for more information on Django's admin error logging.)

You could expand on this concept and create your own EmailBackend that dynamically switches between other Anymail backends—based on properties of the message, or other criteria you set. For example, this gist shows an EmailBackend that checks ESPs' status-page APIs, and automatically falls back to a different ESP when the first one isn't working.

1.6.3 Using Django templates for email

ESP's templating languages and merge capabilities are generally not compatible with each other, which can make it hard to move email templates between them.

But since you're working in Django, you already have access to the extremely-full-featured `Django templating system`. You don't even have to use Django's template syntax: it supports other template languages (like Jinja2).

You're probably already using Django's templating system for your HTML pages, so it can be an easy decision to use it for your email, too.

To compose email using *Django* templates, you can use Django's `render_to_string()` template shortcut to build the body and html.

Example that builds an email from the templates `message_subject.txt`, `message_body.txt` and `message_body.html`:

```
from django.core.mail import EmailMultiAlternatives
from django.template import Context
from django.template.loader import render_to_string

merge_data = {
    'ORDERNO': "12345", 'TRACKINGNO': "1Z987"
}

plaintext_context = Context(autoescape=False)  # HTML escaping not appropriate in␣
↪plaintext
subject = render_to_string("message_subject.txt", merge_data, plaintext_context)
text_body = render_to_string("message_body.txt", merge_data, plaintext_context)
```

(continues on next page)

(continued from previous page)

```
html_body = render_to_string("message_body.html", merge_data)

msg = EmailMultiAlternatives(subject=subject, from_email="store@example.com",
                             to=["customer@example.com"], body=text_body)
msg.attach_alternative(html_body, "text/html")
msg.send()
```

Helpful add-ons

These (third-party) packages can be helpful for building your email in Django:

- django-templated-mail, django-mail-templated, or django-mail-templated-simple for building messages from sets of Django templates.
- premailer for inlining css before sending
- BeautifulSoup, lxml, or html2text for auto-generating plaintext from your html

1.6.4 Securing webhooks

If not used carefully, webhooks can create security vulnerabilities in your Django application.

At minimum, you should **use https** and a **shared authentication secret** for your Anymail webhooks. (Really, for *any* webhooks.)

Does this really matter?

Short answer: yes!

Do you allow unauthorized access to your APIs? Would you want someone eavesdropping on API calls? Of course not. Well, a webhook is just another API.

Think about the data your ESP sends and what your app does with it. If your webhooks aren't secured, an attacker could...

- accumulate a list of your customers' email addresses
- fake bounces and spam reports, so you block valid user emails
- see the full contents of email from your users
- convincingly forge incoming mail, tricking your app into publishing spam or acting on falsified commands
- overwhelm your DB with garbage data (do you store tracking info? incoming attachments?)

... or worse. Why take a chance?

Use https

For security, your Django site must use https. The webhook URLs you give your ESP need to start with *https* (not *http*).

Without https, the data your ESP sends your webhooks is exposed in transit. This can include your customers' email addresses, the contents of messages you receive through your ESP, the shared secret used to authorize calls to your webhooks (described in the next section), and other data you'd probably like to keep private.

Configuring https is beyond the scope of Anymail, but there are many good tutorials on the web. If you've previously dismissed https as too expensive or too complicated, please take another look. Free https certificates are available from Let's Encrypt, and many hosting providers now offer easy https configuration using Let's Encrypt or their own no-cost option.

If you aren't able to use https on your Django site, then you should not set up your ESP's webhooks.

Use a shared authentication secret

A webhook is an ordinary URL—anyone can post anything to it. To avoid receiving random (or malicious) data in your webhook, you should use a shared random secret that your ESP can present with webhook data, to prove the post is coming from your ESP.

Most ESPs recommend using HTTP basic authentication as this shared secret. Anymail includes support for this, via the `ANYMAIL_WEBHOOK_SECRET` setting. Basic usage is covered in the *webhooks configuration* docs.

If something posts to your webhooks without the required shared secret as basic auth in the HTTP_AUTHORIZATION header, Anymail will raise an `AnymailWebhookValidationFailure` error, which is a subclass of Django's `SuspiciousOperation`. This will result in an HTTP 400 response, without further processing the data or calling your signal receiver function.

In addition to a single "random:random" string, you can give a list of authentication strings. Anymail will permit webhook calls that match any of the authentication strings:

```
ANYMAIL = {
    ...
    'WEBHOOK_SECRET': [
        'abcdefghijklmnop:qrstuvwxyz0123456789',
        'ZYXWVUTSRQPONMLK:JIHGFEDCBA9876543210',
    ],
}
```

This facilitates credential rotation: first, append a new authentication string to the list, and deploy your Django site. Then, update the webhook URLs at your ESP to use the new authentication. Finally, remove the old (now unused) authentication string from the list and re-deploy.

Warning: If your webhook URLs don't use https, this shared authentication secret won't stay secret, defeating its purpose.

Signed webhooks

Some ESPs implement webhook signing, which is another method of verifying the webhook data came from your ESP. Anymail will verify these signatures for ESPs that support them. See the docs for your *specific ESP* for more details and configuration that may be required.

Even with signed webhooks, it doesn't hurt to also use a shared secret.

Additional steps

Webhooks aren't unique to Anymail or to ESPs. They're used for many different types of inter-site communication, and you can find additional recommendations for improving webhook security on the web.

For example, you might consider:

- Tracking *event_id*, to avoid accidental double-processing of the same events (or replay attacks)

- Checking the webhook's *timestamp* is reasonably close the current time
- Configuring your firewall to reject webhook calls that come from somewhere other than your ESP's documented IP addresses (if your ESP provides this information)
- Rate-limiting webhook calls in your web server or using something like django-ratelimit

But you should start with using https and a random shared secret via HTTP auth.

1.6.5 Testing your app

Django's own test runner makes sure your test cases don't send email, by loading a dummy EmailBackend that accumulates messages in memory rather than sending them. That works just fine with Anymail.

Anymail also includes its own "test" EmailBackend. This is intended primarily for Anymail's own internal tests, but you may find it useful for some of your test cases, too:

- Like Django's locmem EmailBackend, Anymail's test EmailBackend collects sent messages in `django.core.mail.outbox`. Django clears the outbox automatically between test cases. See email testing tools in the Django docs for more information.
- Unlike the locmem backend, Anymail's test backend processes the messages as though they would be sent by a generic ESP. This means every sent EmailMessage will end up with an *anymail_status* attribute after sending, and some common problems like malformed addresses may be detected. (But no ESP-specific checks are run.)
- Anymail's test backend also adds an `anymail_send_params` attribute to each EmailMessage as it sends it. This is a dict of the actual params that would be used to send the message, including both Anymail-specific attributes from the EmailMessage and options that would come from Anymail settings defaults.

Here's an example:

```
from django.core import mail
from django.test import TestCase
from django.test.utils import override_settings

@override_settings(EMAIL_BACKEND='anymail.backends.test.EmailBackend')
class SignupTestCase(TestCase):
    # Assume our app has a signup view that accepts an email address...
    def test_sends_confirmation_email(self):
        self.client.post("/account/signup/", {"email": "user@example.com"})

        # Test that one message was sent:
        self.assertEqual(len(mail.outbox), 1)

        # Verify attributes of the EmailMessage that was sent:
        self.assertEqual(mail.outbox[0].to, ["user@example.com"])
        self.assertEqual(mail.outbox[0].tags, ["confirmation"])  # an Anymail custom␣
↪attr

        # Or verify the Anymail params, including any merged settings defaults:
        self.assertTrue(mail.outbox[0].anymail_send_params["track_clicks"])
```

1.6.6 Batch send performance

If you are sending batches of hundreds of emails at a time, you can improve performance slightly by reusing a single HTTP connection to your ESP's API, rather than creating (and tearing down) a new connection for each message.

Most Anymail EmailBackends automatically reuse their HTTP connections when used with Django's batch-sending functions `send_mass_mail()` or `connection.send_messages()`. See Sending multiple emails in the Django docs for more info and an example.

(The exception is when Anymail wraps an ESP's official Python package, and that package doesn't support connection reuse. Django's batch-sending functions will still work, but will incur the overhead of creating a separate connection for each message sent. Currently, only SparkPost has this limitation.)

If you need even more performance, you may want to consider your ESP's batch-sending features. When supported by your ESP, Anymail can send multiple messages with a single API call. See *Batch sending with merge data* for details, and be sure to check the *ESP-specific info* because batch sending capabilities vary significantly between ESPs.

1.7 Help

1.7.1 Troubleshooting

If Anymail's not behaving like you expect, these troubleshooting tips can often help you pinpoint the problem...

Check the error message

Look for an Anymail error message in your console (running Django in dev mode) or in your server error logs. If you see something like "invalid API key" or "invalid email address", that's often a big first step toward being able to solve the problem.

Check your ESPs API logs

Most ESPs offer some sort of API activity log in their dashboards. Check their logs to see if the data you thought you were sending actually made it to your ESP, and if they recorded any errors there.

Double-check common issues

- Did you add any required settings for your ESP to the `ANYMAIL` dict in your settings.py? (E.g., `"SENDGRID_API_KEY"` for SendGrid.) See *Supported ESPs*.
- Did you add `'anymail'` to the list of `INSTALLED_APPS` in settings.py?
- Are you using a valid from address? Django's default is "webmaster@localhost", which most ESPs reject. Either specify the `from_email` explicitly on every message you send, or add `DEFAULT_FROM_EMAIL` to your settings.py.

Try it without Anymail

If you think Anymail might be causing the problem, try switching your `EMAIL_BACKEND` setting to Django's File backend and then running your email-sending code again. If that causes errors, you'll know the issue is somewhere other than Anymail. And you can look through the `EMAIL_FILE_PATH` file contents afterward to see if you're generating the email you want.

1.7.2 Support

If you've gone through the troubleshooting above and still aren't sure what's wrong, the Anymail community is happy to help. Anymail is supported and maintained by the people who use it—like you! (The vast majority of Anymail contributors volunteer their time, and are not employees of any ESP.)

Here's how to contact the Anymail community:

"How do I...?"

If the *Search docs* box on the left doesn't find an answer, ask a question on Stack Overflow and tag it "django-anymail".

"I'm getting an error or unexpected behavior…"

Either ask a question on Stack Overflow tagged "django-anymail" or open a GitHub issue. (But please don't raise the same issue in both places.)

Be sure to include:

- which ESP you're using (Mailgun, SendGrid, etc.)
- what versions of Anymail, Django, and Python you're running
- the relevant portions of your code and settings
- the text of any error messages
- any exception stack traces

and any other info you obtained from *troubleshooting*, such as what you found in your ESP's activity log.

"I found a bug…"

Open a GitHub issue. Be sure to include the information listed above. (And if you know what the problem is, we always welcome *contributions* with a fix!)

"I found a security issue!"

Contact the Anymail maintainers by emailing *security<AT>anymail<DOT>info*. (Please don't open a GitHub issue or post publicly about potential security problems.)

"Could Anymail support this ESP or feature…?"

If there's already a GitHub issue open, express your support using GitHub's thumbs up reaction. If not, open a new issue. Either way, be sure to add a comment if you're able to help with development or testing.

1.8 Contributing

Anymail is maintained by its users. Your contributions are encouraged!

The Anymail source code is on GitHub.

1.8.1 Contributors

See AUTHORS.txt for a list of some of the people who have helped improve Anymail.

Anymail evolved from the Djrill project. Special thanks to the folks from brack3t who developed the original version of Djrill.

1.8.2 Bugs

You can report problems or request features in Anymail's GitHub issue tracker. (For a security-related issue that should not be disclosed publicly, instead email Anymail's maintainers at security<AT>anymail<DOT>info.)

We also have some *Troubleshooting* information that may be helpful.

1.8.3 Pull requests

Pull requests are always welcome to fix bugs and improve support for ESP and Django features.

- Please include test cases.
- We try to follow the Django coding style (basically, **PEP 8** with longer lines OK).
- By submitting a pull request, you're agreeing to release your changes under under the same BSD license as the rest of this project.
- Documentation is appreciated, but not required. (Please don't let missing or incomplete documentation keep you from contributing code.)

1.8.4 Testing

Anymail is tested on Travis CI against several combinations of Django and Python versions. Tests are run at least once a week, to check whether ESP APIs and other dependencies have changed out from under Anymail.

For local development, the recommended test command is `tox -e django21-py36-all,django111-py27-all,lint`, which tests a representative combination of Python and Django versions. It also runs flake8 and other code-style checkers. Some other test options are covered below, but using this tox command catches most problems, and is a good pre-pull-request check.

Most of the included tests verify that Anymail constructs the expected ESP API calls, without actually calling the ESP's API or sending any email. So these tests don't require API keys, but they *do* require mock and all ESP-specific package requirements.

To run the tests, you can:

```
$ python setup.py test  # (also installs test dependencies if needed)
```

Or:

```
$ pip install mock boto3 sparkpost  # install test dependencies
$ python runtests.py

## this command can also run just a few test cases, e.g.:
$ python runtests.py tests.test_mailgun_backend tests.test_mailgun_webhooks
```

Or to test against multiple versions of Python and Django all at once, use tox. You'll need at least Python 2.7 and Python 3.6 available. (If your system doesn't come with those, pyenv is a helpful way to install and manage multiple Python versions.)

```
$ pip install tox  # (if you haven't already)
$ tox -e django21-py36-all,django111-py27-all,lint  # test recommended␣
↪environments

## you can also run just some test cases, e.g.:
$ tox -e django21-py36-all,django111-py27-all tests.test_mailgun_backend␣
↪tests.test_utils

## to test more Python/Django versions:
$ tox  # ALL 20+ envs! (grab a coffee, or use `detox` to run tests in␣
↪parallel)
$ tox --skip-missing-interpreters  # if some Python versions aren't installed
```

In addition to the mocked tests, Anymail has integration tests which *do* call live ESP APIs. These tests are normally skipped; to run them, set environment variables with the necessary API keys or other settings. For example:

```
$ export MAILGUN_TEST_API_KEY='your-Mailgun-API-key'
$ export MAILGUN_TEST_DOMAIN='mail.example.com'  # sending domain for that␣
↪API key
$ tox -e django21-py36-all tests.test_mailgun_integration
```

Check the `*_integration_tests.py` files in the tests source to see which variables are required for each ESP. Depending on the supported features, the integration tests for a particular ESP send around 5-15 individual messages. For ESPs that don't offer a sandbox, these will be real sends charged to your account (again, see the notes in each test case). Be sure to specify a particular testenv with tox's `-e` option, or tox may repeat the tests for all 20+ supported combinations of Python and Django, sending hundreds of messages.

1.8.5 Documentation

As noted above, Anymail welcomes pull requests with missing or incomplete documentation. (Code without docs is better than no contribution at all.) But documentation—even needing edits—is always appreciated, as are pull requests simply to improve the docs themselves.

Like many Python packages, Anymail's docs use Sphinx. If you've never worked with Sphinx or reStructuredText, Django's Writing Documentation can get you started.

It's easiest to build Anymail's docs using tox:

```
$ pip install tox  # (if you haven't already)
$ tox -e docs  # build the docs using Sphinx
```

You can run Python's simple HTTP server to view them:

```
$ (cd .tox/docs/_html; python3 -m http.server 8123 --bind 127.0.0.1)
```

. . . and then open http://localhost:8123/ in a browser. Leave the server running, and just re-run the tox command and refresh your browser as you make changes.

If you've edited the main README.rst, you can preview an approximation of what will end up on PyPI at http://localhost:8123/readme.html.

Anymail's Sphinx conf sets up a few enhancements you can use in the docs:

- Loads intersphinx mappings for Python 3, Django (stable), and Requests. Docs can refer to things like `:ref:`django:topics-testing-email`` or `:class:`django.core.mail.EmailMessage``.
- Supports much of Django's added markup, notably `:setting:` for documenting or referencing Django and Anymail settings.
- Allows linking to Python packages with `:pypi:`package-name`` (via extlinks).

1.9 Changelog

Anymail releases follow semantic versioning. Among other things, this means that minor updates (1.x to 1.y) should always be backwards-compatible, and breaking changes will always increment the major version number (1.x to 2.0).

1.9.1 Release history

v7.0

2019-09-07

Breaking changes

- **SendinBlue templates:** Support Sendinblue's new (ESP stored) Django templates and new API for template sending. This removes most of the odd limitations in the older (now-deprecated) SendinBlue template send API, but involves two breaking changes:
  - You *must* convert each old Sendinblue template to the new language as you upgrade to Anymail v7.0, or certain features may be silently ignored on template sends (notably `reply_to` and recipient display names).
  - Sendinblue's API no longer supports sending attachments when using templates.

  Ordinary, non-template sending is not affected by these changes. See docs for more info and alternatives. (Thanks @Thorbenl.)

Features

- **Mailgun:** Support Mailgun's new (ESP stored) handlebars templates via `template_id`. See docs. (Thanks @anstosa.)
- **SendinBlue:** Support multiple `tags`. (Thanks @Thorbenl.)

Other

- **Mailgun:** Disable Anymail's workaround for a Requests/urllib3 issue with non-ASCII attachment filenames when a newer version of urllib3–which fixes the problem–is installed. (Workaround was added in Anymail v4.3; fix appears in urllib3 v1.25.)

v6.1

2019-07-07

Features

- **Mailgun:** Add new `MAILGUN_WEBHOOK_SIGNING_KEY` setting for verifying tracking and inbound webhook calls. Mailgun's webhook signing key can become different from your `MAILGUN_API_KEY` if you have ever rotated either key. See docs. (More in #153. Thanks to @dominik-lekse for reporting the problem and Mailgun's @mbk-ok for identifying the cause.)

v6.0.1

2019-05-19

Fixes

- Support using `AnymailMessage` with django-mailer and similar packages that pickle messages. (See #147. Thanks to @ewingrj for identifying the problem.)
- Fix UnicodeEncodeError error while reporting invalid email address on Python 2.7. (See #148. Thanks to @fdemmer for reporting the problem.)

v6.0

2019-02-23

Breaking changes

- **Postmark:** Anymail's `message.anymail_status.recipients[email]` no longer lowercases the recipient's email address. For consistency with other ESPs, it now uses the recipient email with whatever case was used in the sent message. If your code is doing something like `message.anymail_status.recipients[email.lower()]`, you should remove the `.lower()`
- **SendGrid:** In batch sends, Anymail's SendGrid backend now assigns a separate `message_id` for each "to" recipient, rather than sharing a single id for all recipients. This improves accuracy of tracking and statistics (and matches the behavior of many other ESPs).

  If your code uses batch sending (merge_data with multiple to-addresses) and checks `message.anymail_status.message_id` after sending, that value will now be a *set* of ids. You can obtain each recipient's individual message_id with `message.anymail_status.recipients[to_email].message_id`. See docs.

Features

- Add new `merge_metadata` option for providing per-recipient metadata in batch sends. Available for all supported ESPs *except* Amazon SES and SendinBlue. See docs. (Thanks @janneThoft for the idea and SendGrid implementation.)
- **Mailjet:** Remove limitation on using `cc` or `bcc` together with `merge_data`.

Fixes

- **Mailgun:** Better error message for invalid sender domains (that caused a cryptic "Mailgun API response 200: OK Mailgun Magnificent API" error in earlier releases).
- **Postmark:** Don't error if a message is sent with only Cc and/or Bcc recipients (but no To addresses). Also, `message.anymail_status.recipients[email]` now includes send status for Cc and Bcc recipients. (Thanks to @ailionx for reporting the error.)
- **SendGrid:** With legacy templates, stop (ab)using "sections" for merge_global_data. This avoids potential conflicts with a template's own use of SendGrid section tags.

v5.0

2018-11-07

Breaking changes

- **Mailgun:** Anymail's status tracking webhooks now report Mailgun "temporary failure" events as Anymail's normalized "deferred" `event_type`. (Previously they were reported as "bounced", lumping them in with permanent failures.) The new behavior is consistent with how Anymail handles other ESP's tracking notifications. In the unlikely case your code depended on "temporary failure" showing up as "bounced" you will need to update it. (Thanks @costela.)

Features

- **Postmark:** Allow either template alias (string) or numeric template id for Anymail's `template_id` when sending with Postmark templates.

Fixes

- **Mailgun:** Improve error reporting when an inbound route is accidentally pointed at Anymail's tracking webhook url or vice versa.

v4.3

2018-10-11

Features

- Treat MIME attachments that have a *Content-ID* but no explicit *Content-Disposition* header as inline, matching the behavior of many email clients. For maximum compatibility, you should always set both (or use Anymail's inline helper functions). (Thanks @costela.)

Fixes

- **Mailgun:** Raise `AnymailUnsupportedFeature` error when attempting to send an attachment without a filename (or inline attachment without a *Content-ID*), because Mailgun silently drops these attachments from the sent message. (See docs. Thanks @costela for identifying this undocumented Mailgun API limitation.)
- **Mailgun:** Fix problem where attachments with non-ASCII filenames would be lost. (Works around Requests/urllib3 issue encoding multipart/form-data filenames in a way that isn't RFC 7578 compliant. Thanks to @decibyte for catching the problem.)

Other

- Add (undocumented) DEBUG_API_REQUESTS Anymail setting. When enabled, prints raw API request and response during send. Currently implemented only for Requests-based backends (all but Amazon SES and SparkPost). Because this can expose API keys and other sensitive info in log files, it should not be used in production.

v4.2

2018-09-07

Features

- **Postmark:** Support per-recipient template `merge_data` and batch sending. (Batch sending can be used with or without a template. See docs.)

Fixes

- **Postmark:** When using `template_id`, ignore empty subject and body. (Postmark issues an error if Django's default empty strings are used with template sends.)

v4.1

2018-08-27

Features

- **SendGrid:** Support both new "dynamic" and original "legacy" transactional templates. (See docs.)
- **SendGrid:** Allow merging `esp_extra["personalizations"]` dict into other message-derived personalizations. (See docs.)

v4.0

2018-08-19

Breaking changes

- Drop support for Django versions older than Django 1.11. (For compatibility back to Django 1.8, stay on the Anymail *v3.0* extended support branch.)
- **SendGrid:** Remove the legacy SendGrid *v2* EmailBackend. (Anymail's default since v0.8 has been SendGrid's newer v3 API.) If your settings.py `EMAIL_BACKEND` still references "sendgrid_v2," you must upgrade to v3.

Features

- **Mailgun:** Add support for new Mailgun webhooks. (Mailgun's original "legacy webhook" format is also still supported. See docs.)
- **Mailgun:** Document how to use new European region. (This works in earlier Anymail versions, too.)
- **Postmark:** Add support for Anymail's normalized `metadata` in sending and webhooks.

Fixes

- Avoid problems with Gmail blocking messages that have inline attachments, when sent from a machine whose local hostname ends in *.com*. Change Anymail's `attach_inline_image()` default *Content-ID* domain to the literal text "inline" (rather than Python's default of the local hostname), to work around a limitation of some ESP APIs that don't permit distinct content ID and attachment filenames (Mailgun, Mailjet, Mandrill and SparkPost). See #112 for more details.

- **Amazon SES:** Work around an Amazon SES bug that can corrupt non-ASCII message bodies if you are using SES's open or click tracking. (See #115 for more details. Thanks to @varche1 for isolating the specific conditions that trigger the bug.)

Other

- Maintain changelog in the repository itself (rather than in GitHub release notes).
- Test against released versions of Python 3.7 and Django 2.1.

v3.0

2018-05-30

This is an extended support release. Anymail v3.x will receive security updates and fixes for any breaking ESP API changes through at least April, 2019.

Breaking changes

- Drop support for Python 3.3 (see #99).
- **SendGrid:** Fix a problem where Anymail's status tracking webhooks didn't always receive the same `event.message_id` as the sent `message.anymail_status.message_id`, due to unpredictable behavior by SendGrid's API. Anymail now generates a UUID for each sent message and attaches it as a SendGrid custom arg named anymail_id. For most users, this change should be transparent. But it could be a breaking change if you are relying on a specific message_id format, or relying on message_id matching the *Message-ID* mail header or SendGrid's "smtp-id" event field. (More details in the docs; also see #108.) Thanks to @joshkersey for the report and the fix.

Features

- Support Django 2.1 prerelease.

Fixes

- **Mailjet:** Fix tracking webhooks to work correctly when Mailjet "group events" option is disabled (see #106).

Deprecations

- This will be the last Anymail release to support Django 1.8, 1.9, and 1.10 (see #110).
- This will be the last Anymail release to support the legacy SendGrid v2 EmailBackend (see #111). (SendGrid's newer v3 API has been the default since Anymail v0.8.)

If these deprecations affect you and you cannot upgrade, set your requirements to `django-anymail~=3.0` (a "compatible release" specifier, equivalent to `>=3.0,==3.*`).

v2.2

2018-04-16

Fixes

- Fix a breaking change accidentally introduced in v2.1: The boto3 package is no longer required if you aren't using Amazon SES.

v2.1

2018-04-11

NOTE: v2.1 accidentally introduced a **breaking change:** enabling Anymail webhooks with `include('anymail.urls')` causes an error if boto3 is not installed, even if you aren't using Amazon SES. This is fixed in v2.2.

Features

- **Amazon SES:** Add support for this ESP (docs).
- **SparkPost:** Add SPARKPOST_API_URL setting to support SparkPost EU and SparkPost Enterprise (docs).
- **Postmark:** Update for Postmark "modular webhooks." This should not impact client code. (Also, older versions of Anymail will still work correctly with Postmark's webhook changes.)

Fixes

- **Inbound:** Fix several issues with inbound messages, particularly around non-ASCII headers and body content. Add workarounds for some limitations in older Python email packages.

Other

- Use tox to manage Anymail test environments (see contributor docs).

Deprecations

- This will be the last Anymail release to support Python 3.3. See #99 for more information.

v2.0

2018-03-08

Breaking changes

- Drop support for deprecated WEBHOOK_AUTHORIZATION setting. If you are using webhooks and still have this Anymail setting, you must rename it to WEBHOOK_SECRET. See the *v1.4* release notes.
- Handle *Reply-To, From,* and *To* in EmailMessage `extra_headers` the same as Django's SMTP EmailBackend if supported by your ESP, otherwise raise an unsupported feature error. Fixes the SparkPost backend to be consistent with other backends if both `headers["Reply-To"]` and `reply_to` are set on the same message. If you are setting a message's `headers["From"]` or `headers["To"]` (neither is common), the new behavior is likely a breaking change. See docs and #91.

- Treat EmailMessage `extra_headers` keys as case-*insensitive* in all backends, for consistency with each other (and email specs). If you are specifying duplicate headers whose names differ only in case, this may be a breaking change. See docs.

Features

- **SendinBlue:** Add support for this ESP (docs). Thanks to @RignonNoel for the implementation.
- Add EmailMessage `envelope_sender` attribute, which can adjust the message's *Return-Path* if supported by your ESP (docs).
- Add universal wheel to PyPI releases for faster installation.

Other

- Update setup.py metadata, clean up implementation. (Hadn't really been touched since original Djrill version.)
- Prep for Python 3.7.

v1.4

2018-02-08

Security

- Fix a low severity security issue affecting Anymail v0.2–v1.3: rename setting WEBHOOK_AUTHORIZATION to WEBHOOK_SECRET to prevent inclusion in Django error reporting. (CVE-2018-1000089)

More information

Django error reporting includes the value of your Anymail WEBHOOK_AUTHORIZATION setting. In a properly-configured deployment, this should not be cause for concern. But if you have somehow exposed your Django error reports (e.g., by mis-deploying with DEBUG=True or by sending error reports through insecure channels), anyone who gains access to those reports could discover your webhook shared secret. An attacker could use this to post fabricated or malicious Anymail tracking/inbound events to your app, if you are using those Anymail features.

The fix renames Anymail's webhook shared secret setting so that Django's error reporting mechanism will sanitize it.

If you are using Anymail's event tracking and/or inbound webhooks, you should upgrade to this release and change "WEBHOOK_AUTHORIZATION" to "WEBHOOK_SECRET" in the ANYMAIL section of your settings.py. You may also want to rotate the shared secret value, particularly if you have ever exposed your Django error reports to untrusted individuals.

If you are only using Anymail's EmailBackends for sending email and have not set up Anymail's webhooks, this issue does not affect you.

The old WEBHOOK_AUTHORIZATION setting is still allowed in this release, but will issue a system-check warning when running most Django management commands. It will be removed completely in a near-future release, as a breaking change.

Thanks to Charlie DeTar (@yourcelf) for responsibly reporting this security issue through private channels.

v1.3

2018-02-02

Security

- v1.3 includes the v1.2.1 security fix released at the same time. Please review the *v1.2.1* release notes, below, if you are using Anymail's tracking webhooks.

Features

- **Inbound handling:** Add normalized inbound message event, signal, and webhooks for all supported ESPs. (See new Receiving mail docs.) This hasn't been through much real-world testing yet; bug reports and feedback are very welcome.
- **API network timeouts:** For Requests-based backends (all but SparkPost), use a default timeout of 30 seconds for all ESP API calls, to avoid stalling forever on a bad connection. Add a REQUESTS_TIMEOUT Anymail setting to override. (See #80.)
- **Test backend improvements:** Generate unique tracking `message_id` when using the test backend; add console backend for use in development. (See #85.)

v1.2.1

2018-02-02

Security

- Fix a **moderate severity** security issue affecting Anymail v0.2–v1.2: prevent timing attack on WEBHOOK_AUTHORIZATION secret. (CVE-2018-6596)

More information

If you are using Anymail's tracking webhooks, you should upgrade to this release, and you may want to rotate to a new WEBHOOK_AUTHORIZATION shared secret (see docs). You should definitely change your webhook auth if your logs indicate attempted exploit.

(If you are only sending email using an Anymail EmailBackend, and have not set up Anymail's event tracking webhooks, this issue does not affect you.)

Anymail's webhook validation was vulnerable to a timing attack. A remote attacker could use this to obtain your WEBHOOK_AUTHORIZATION shared secret, potentially allowing them to post fabricated or malicious email tracking events to your app.

There have not been any reports of attempted exploit. (The vulnerability was discovered through code review.) Attempts would be visible in HTTP logs as a very large number of 400 responses on Anymail's webhook urls (by default "/anymail/*esp_name*/tracking/"), and in Python error monitoring as a very large number of AnymailWebhookValidationFailure exceptions.

v1.2

2017-11-02

Features

- **Postmark:** Support new click webhook in normalized tracking events

v1.1

2017-10-28

Fixes

- **Mailgun:** Support metadata in opened/clicked/unsubscribed tracking webhooks, and fix potential problems if metadata keys collided with Mailgun event parameter names. (See #76, #77)

Other

- Rework Anymail's ParsedEmail class and rename to EmailAddress to align it with similar functionality in the Python 3.6 email package, in preparation for future inbound support. ParsedEmail was not documented for use outside Anymail's internals (so this change does not bump the semver major version), but if you were using it in an undocumented way you will need to update your code.

v1.0

2017-09-18

It's official: Anymail is no longer "pre-1.0." The API has been stable for many months, and there's no reason not to use Anymail in production.

Breaking changes

- There are no *new* breaking changes in the 1.0 release, but a breaking change introduced several months ago in v0.8 is now strictly enforced. If you still have an EMAIL_BACKEND setting that looks like "anymail.backends.*espname*.*EspName*Backend", you'll need to change it to just "anymail.backends.*espname*.EmailBackend". (Earlier versions had issued a DeprecationWarning. See the *v0.8* release notes.)

Features

- Clean up and document Anymail's Test EmailBackend
- Add notes on handling transient ESP errors and improving batch send performance
- **SendGrid:** handle Python 2 `long` integers in metadata and extra headers

v1.0.rc0

2017-09-09

Breaking changes

- **All backends:** The old *EspName*Backend names that were deprecated in v0.8 have been removed. Attempting to use the old names will now fail, rather than issue a DeprecationWarning. See the *v0.8* release notes.

Features

- Anymail's Test EmailBackend is now documented (and cleaned up)

v0.11.1

2017-07-24

Fixes

- **Mailjet:** Correct settings docs.

v0.11

2017-07-13

Features

- **Mailjet:** Add support for this ESP. Thanks to @Lekensteyn and @calvin. (Docs)
- In webhook handlers, AnymailTrackingEvent.metadata now defaults to `{}`, and .tags defaults to `[]`, if the ESP does not supply these fields with the event. (See #67.)

v0.10

2017-05-22

Features

- **Mailgun, SparkPost:** Support multiple from addresses, as a comma-separated `from_email` string. (*Not* a list of strings, like the recipient fields.) RFC-5322 allows multiple from email addresses, and these two ESPs support it. Though as a practical matter, multiple from emails are either ignored or treated as a spam signal by receiving mail handlers. (See #60.)

Fixes

- Fix crash sending forwarded email messages as attachments. (See #59.)
- **Mailgun:** Fix webhook crash on bounces from some receiving mail handlers. (See #62.)
- Improve recipient-parsing error messages and consistency with Django's SMTP backend. In particular, Django (and now Anymail) allows multiple, comma-separated email addresses in a single recipient string.

v0.9

2017-04-04

Breaking changes

- **Mandrill, Postmark:** Normalize soft-bounce webhook events to event_type 'bounced' (rather than 'deferred').

Features

- Officially support released Django 1.11, including under Python 3.6.

v0.8

2017-02-02

Breaking changes

- **All backends:** Rename all Anymail backends to just `EmailBackend`, matching Django's naming convention. E.g., you should update: `EMAIL_BACKEND = "anymail.backends.mailgun.MailgunBackend" # old` to: `EMAIL_BACKEND = "anymail.backends.mailgun.EmailBackend" # new`

  The old names still work, but will issue a DeprecationWarning and will be removed in some future release (Apologies for this change; the old naming was a holdover from Djrill, and I wanted to establish consistency with other Django EmailBackends before Anymail 1.0. See #49.)

- **SendGrid:** Update SendGrid backend to their newer Web API v3. This should be a transparent change for most projects. Exceptions: if you use SendGrid username/password auth, Anymail's `esp_extra` with "x-smtpapi", or multiple Reply-To addresses, please review the porting notes.

  The SendGrid v2 EmailBackend remains available if you prefer it, but is no longer the default.

Features

- Test on Django 1.11 prerelease, including under Python 3.6.

Fixes

- **Mandrill:** Fix bug in webhook signature validation when using basic auth via the WEBHOOK_AUTHORIZATION setting. (If you were using the MANDRILL_WEBHOOK_URL setting to work around this problem, you should be able to remove it. See #48.)

v0.7

2016-12-30

Breaking changes

- Fix a long-standing bug validating email addresses. If an address has a display name containing a comma or parentheses, RFC-5322 *requires* double-quotes around the display name (`'"Widgets, Inc." <widgets@example.com>'`). Anymail now raises a new `AnymailInvalidAddress` error for misquoted display names and other malformed addresses. (Previously, it silently truncated the address, leading to

obscure exceptions or unexpected behavior. If you were unintentionally relying on that buggy behavior, this may be a breaking change. See #44.) In general, it's safest to always use double-quotes around all display names.

Features

- **Postmark:** Support Postmark's new message delivery event in Anymail normalized tracking webhook. (Update your Postmark config to enable the new event. See docs.)
- Handle virtually all uses of Django lazy translation strings as EmailMessage properties. (In earlier releases, these could sometimes lead to obscure exceptions or unexpected behavior with some ESPs. See #34.)
- **Mandrill:** Simplify and document two-phase process for setting up Mandrill webhooks (docs).

v0.6.1

2016-11-01

Fixes

- **Mailgun, Mandrill:** Support older Python 2.7.x versions in webhook validation (#39; thanks @sebbacon).
- **Postmark:** Handle older-style 'Reply-To' in EmailMessage `headers` (#41).

v0.6

2016-10-25

Breaking changes

- **SendGrid:** Fix missing html or text template body when using `template_id` with an empty Django EmailMessage body. In the (extremely-unlikely) case you were relying on the earlier quirky behavior to *not* send your saved html or text template, you may want to verify that your SendGrid templates have matching html and text. (docs – also see #32.)

Features

- **Postmark:** Add support for `track_clicks` (docs)
- Initialize AnymailMessage.anymail_status to empty status, rather than None; clarify docs around `anymail_status` availability (docs)

v0.5

2016-08-22

Features

- **Mailgun:** Add MAILGUN_SENDER_DOMAIN setting. (docs)

v0.4.2

2016-06-24

Fixes

- **SparkPost:** Fix API error "Both content object and template_id are specified" when using `template_id` (#24).

v0.4.1

2016-06-23

Features

- **SparkPost:** Add support for this ESP. (docs)
- Test with Django 1.10 beta
- Requests-based backends (all but SparkPost) now raise AnymailRequestsAPIError for any requests.RequestException, for consistency and proper fail_silently behavior. (The exception will also be a subclass of the original RequestException, so no changes are required to existing code looking for specific requests failures.)

v0.4

(not released)

v0.3.1

2016-05-18

Fixes

- **SendGrid:** Fix API error that `to` is required when using `merge_data` (see #14; thanks @lewistaylor).

v0.3

2016-05-13

Features

- Add support for ESP stored templates and batch sending/merge. Exact capabilities vary widely by ESP – be sure to read the notes for your ESP. (docs)
- Add pre_send and post_send signals. docs
- **Mandrill:** add support for esp_extra; deprecate Mandrill-specific message attributes left over from Djrill. See migrating from Djrill.

v0.2

2016-04-30

Breaking changes

- **Mailgun:** eliminate automatic JSON encoding of complex metadata values like lists and dicts. (Was based on misreading of Mailgun docs; behavior now matches metadata handling for all other ESPs.)
- **Mandrill:** remove obsolete wehook views and signal inherited from Djrill. See Djrill migration notes if you were relying on that code.

Features

- Add support for ESP event-tracking webhooks, including normalized AnymailTrackingEvent. (docs)
- Allow get_connection kwargs overrides of most settings for individual backend instances. Can be useful for, e.g., working with multiple SendGrid subusers. (docs)
- **SendGrid:** Add SENDGRID_GENERATE_MESSAGE_ID setting to control workarounds for ensuring unique tracking ID on SendGrid messages/events (default enabled). docs
- **SendGrid:** improve handling of 'filters' in esp_extra, making it easier to mix custom SendGrid app filter settings with Anymail normalized message options.

Other

- Drop pre-Django 1.8 test code. (Wasn't being used, as Anymail requires Django 1.8+.)
- **Mandrill:** note limited support in docs (because integration tests no longer available).

v0.1

2016-03-14

Although this is an early release, it provides functional Django EmailBackends and passes integration tests with all supported ESPs (Mailgun, Mandrill, Postmark, SendGrid).

It has (obviously) not yet undergone extensive real-world testing, and you are encouraged to monitor it carefully if you choose to use it in production. Please report bugs and problems here in GitHub.

Features

- **Postmark:** Add support for this ESP.
- **SendGrid:** Add support for username/password auth.
- Simplified install: no need to name the ESP (`pip install django-anymail` – not `... django-anymail[mailgun]`)

0.1.dev2

2016-03-12

Features

- **SendGrid:** Add support for this ESP.
- Add attach_inline_image_file helper

Fixes

- Change inline-attachment handling to look for `Content-Disposition: inline`, and to preserve filenames where supported by the ESP.

0.1.dev1

2016-03-10

Features

- **Mailgun, Mandrill:** initial supported ESPs.
- Initial docs

1.10 Anymail documentation privacy

Anymail's documentation site at anymail.readthedocs.io is hosted by **Read the Docs**. Please see the Read the Docs Privacy Policy for more about what information Read the Docs collects and how they use it.

Separately, Anymail's maintainers have configured **Google Analytics** third-party tracking on this documentation site. We (Anymail's maintainers) use this analytics data to better understand how these docs are used, for the purpose of improving the content. Google Analytics helps us answer questions like:

- what docs pages are most and least viewed
- what terms people search for in the documentation
- what paths readers (in general) tend to take through the docs

But we're *not* able to identify any particular person or track individual behavior. Anymail's maintainers *do not* collect or have access to any personally identifiable (or even *potentially* personally identifiable) information about visitors to this documentation site.

We also use Google Analytics to collect feedback from the "Is this page helpful?" box at the bottom of the page. Please do not include any personally-identifiable information in suggestions you submit through this form. (If you would like to contact Anymail's maintainers, see *Support*.)

Anymail's maintainers have *not* connected our Google Analytics implementation to any Google Advertising Services. (Incidentally, we're not involved with the ads you may see here. Those come from—and support—Read the Docs under their ethical ads model.)

The developer audience for Anymail's docs is likely already familiar with site analytics, tracking cookies, and related concepts. To learn more about how Google Analytics uses **cookies** and how to **opt out** of analytics tracking, see the "Information for Visitors of Sites and Apps Using Google Analytics" section of Google's Safeguarding your data document.

Questions about privacy and information practices related to this Anymail documentation site can be emailed to *privacy<at>anymail<dot>info*. (This is not an appropriate contact for questions about *using* Anymail; see *Help* if you need assistance with your code.)

Python Module Index

a

Index

A

C

D

E

F

G

H

I

M

P

R

S

T

U