Anymail Documentation

Release 1.4

Anymail contributors (see AUTHORS.txt)

Feb 08, 2018

Contents

1 Documentation **3**
- 1.1 Anymail 1-2-3 . . . 3
- 1.2 Installation and configuration . . . 4
- 1.3 Sending email . . . 8
- 1.4 Receiving mail . . . 26
- 1.5 Supported ESPs . . . 32
- 1.6 Tips, tricks, and advanced usage . . . 56
- 1.7 Troubleshooting . . . 60
- 1.8 Contributing . . . 61
- 1.9 Release notes . . . 62

Python Module Index **63**

Version 1.4

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 Mailgun, Mailjet, Postmark, 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 1.8–2.0 (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
```

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 a `DEFAULT_FROM_EMAIL` in your settings, this is a good time to add one. (Django's default is "webmaster@localhost", which some ESPs will reject.)

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 authorization. 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 authorization, 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. The old name is still allowed in v1.4, but will be removed in a near-future release. You should update your settings.

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's 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.

Additional headers

Anymail passes additional headers to your ESP. (Some ESPs may limit which headers they'll allow.)

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

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`*.)

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*.

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 *`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

# 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. (But be aware the default `domain` can leak your server's local hostname in the resulting email.)

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,
    },
}
```

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

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`* 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='utf-8'*)
Returns the content of the attachment decoded to a `str` in the given charset. (This is generally only appropriate for text or message-type attachments.)

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 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. Your Mailgun API key:

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

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, include `sender_domain` in Anymail's *`esp_extra`* for that message:

```
message = EmailMessage(from_email="marketing@example.com", ...)
message.esp_extra = {"sender_domain": "mail2.example.com"}
```

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

Metadata keys and tracking webhooks Because of the way Mailgun supplies custom data (user-variables) to webhooks, there are a few metadata keys that Anymail cannot reliably retrieve in some tracking events. You should

avoid using "body-plain", "h", "message-headers", "message-id" or "tag" as *metadata* keys if you need to access that metadata from an opened, clicked, or unsubscribed *tracking event* handler.

Batch sending/merge and ESP templates

Mailgun does not offer *ESP stored templates*, so Anymail's *template_id* message attribute is not supported with the Mailgun backend.

Mailgun *does* support *batch sending* with per-recipient merge data. You can refer to Mailgun "recipient variables" in your message subject and body, and supply the values with Anymail's normalized *merge_data* and *merge_global_data* message attributes:

```
message = EmailMessage(
    ...
    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
}
```

Mailgun does not natively support global merge data. Anymail emulates the capability by copying any `merge_global_data` values to each recipient's section in Mailgun's "recipient-variables" API parameter.

See the Mailgun batch sending docs for more information.

Status tracking webhooks

If you are using Anymail's normalized *status tracking*, enter the url in your Mailgun dashboard on the "Webhooks" tab. 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

If you use multiple Mailgun sending domains, you'll need to enter the webhook URLs for each of them, using the selector on the left side of Mailgun's dashboard.

Mailgun implements a limited form of webhook signing, and Anymail will verify these signatures (based on your *MAILGUN_API_KEY* Anymail setting).

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

The event's *esp_event* field will be a Django `QueryDict` object of Mailgun event fields.

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 "Routes" tab in your Mailgun dashboard.)

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 dashboard on the "Domains" tab).

1.5.2 Mailjet

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

New in version 0.11.

Note: Mailjet is developing an improved v3.1 Send API (in public beta as of mid-2017). Once the v3.1 API is released, Anymail will switch to it. 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`*.

Commas in recipient names Mailjet's v3 API does not properly handle commas in recipient display-names *if* your message also uses the `cc` or `bcc` fields. (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 either avoid using `cc` and `bcc`, or strip commas from all recipient names (in `to`, `cc`, *and* `bcc`) before sending.

Merge data not compatible with cc/bcc Mailjet's v3 API is not capable of representing both `cc` or `bcc` fields and *`merge_data`* in the same message. If you attempt to combine them, Anymail will raise an error at send time.

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

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.)

1.5.3 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.

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.

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.

1.5.4 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 metadata Postmark does not support attaching *metadata* to messages.

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.

Batch sending/merge and ESP templates

Postmark 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* message attribute is not supported with the Postmark backend.

To use a Postmark template, set the message's *template_id* to the numeric Postmark "TemplateID" and supply the "TemplateModel" using the *merge_global_data* message attribute:

```
message = EmailMessage(
    ...
    subject=None,  # use template subject
    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 = 80801  # use this Postmark template
message.merge_global_data = {
    'name': "Alice",
    'order_no': "12345",
    'ship_date': "May 15",
    'items': [
        {'product': "Widget", 'price': "9.99"},
        {'product': "Gadget", 'price': "17.99"},
    ],
}
```

Set the EmailMessage's subject to `None` to use the subject from your Postmark template, or supply a subject with the message to override the template value.

See this Postmark blog post on templates for more information.

Status tracking webhooks

If you are using Anymail's normalized *status tracking*, enter the url in your Postmark account settings, under Servers > *your server name* > Settings > Outbound > Webhooks. You should enter this same Anymail tracking URL for all of the "Delivery webhook," "Bounce webhook," and "Opens webhook" (if you want to receive all these types of events):

`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

Anymail doesn't care about the "include bounce content" and "post only on first open" Postmark webhook settings: 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, or open webhook 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.

1.5.5 SendGrid

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

Changed in version 0.8: Earlier Anymail releases used SendGrid's v2 API. If you are upgrading, please review the *porting notes*.

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 Message-ID for messages sent through SendGrid, to facilitate event tracking.

Default `True`. You can set to `False` to disable this behavior. See *Message-ID quirks* below.

SENDGRID_MERGE_FIELD_FORMAT

If you use *merge data*, set this to a `str.format()` formatting string that indicates how merge fields are delimited in your SendGrid 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.

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.

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%%",
        },
    },
}
```

(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 by default generates a new Message-ID for each outgoing message, provides it to SendGrid, and includes it in the *anymail_status* attribute after you send the message.

In later SendGrid API calls, you can match that Message-ID to SendGrid's `smtp-id` event field. (Anymail uses an additional workaround to ensure smtp-id is included in all SendGrid events, even those that aren't documented to include it.)

Anymail will use the domain of the message's `from_email` to generate the Message-ID. (If this isn't desired, you can supply your own Message-ID in the message's `extra_headers`.)

To disable all of these Message-ID workarounds, set *ANYMAIL_SENDGRID_GENERATE_MESSAGE_ID* to False in your settings.

Single Reply-To SendGrid's v3 API only supports a single Reply-To address (and blocks a workaround that was possible with the v2 API).

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)

Batch sending/merge and ESP templates

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

You can use a SendGrid stored template by setting a message's *template_id* to the template's unique id. 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.

```
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 = "5997fcf6-2b9f-484d-acd5-7e9a99f0dc1f"  # SendGrid 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",
}
message.esp_extra = {
    # Tell Anymail this SendGrid template uses "-field-" to refer to merge␣
↪fields.
    # (We could also just set SENDGRID_MERGE_FIELD_FORMAT in our ANYMAIL␣
↪settings.)
    'merge_field_format': "-{}-"
}
```

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 above. (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 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 difficult to move to other ESPs.)

When you supply per-recipient *merge_data*, Anymail automatically changes how it communicates the "to" list to SendGrid, so that 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.)

SendGrid 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`.

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

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.

Upgrading to SendGrid's v3 API

Anymail v0.8 switched to SendGrid's preferred v3 send API. (Earlier Anymail releases used their v2 API.)

For many Anymail projects, this change will be entirely transparent. (Anymail's whole reason for existence is abstracting ESP APIs, so that your own code doesn't need to worry about the details.)

There are three cases where SendGrid has changed features that would require updates to your code:

1. If you are using SendGrid's username/password auth (your settings include *SENDGRID_USERNAME* and *SENDGRID_PASSWORD*), you must switch to an API key. See *SENDGRID_API_KEY*.

   (If you are already using a SendGrid API key with v2, it should work just fine with v3.)

2. If you are using Anymail's *esp_extra* attribute to supply API-specific parameters, the format has changed.

   Search your code for "esp_extra" (e.g., `git grep esp_extra`) to determine whether this affects you. (Anymail's `"merge_field_format"` is unchanged, so if that's the only thing you have in esp_extra, no changes are needed.)

   The new API format is considerably simpler and more logical. See *esp_extra support* below for examples of the new format and a link to relevant SendGrid docs.

   Anymail will raise an error if it detects an attempt to use the v2-only `"x-smtpapi"` settings in esp_extra when sending.

3. If you send messages with multiple Reply-To addresses, SendGrid no longer supports this. (Multiple reply emails in a single message are not common.)

   Anymail will raise an error if you attempt to send a message with multiple Reply-To emails. (You can suppress the error with *ANYMAIL_IGNORE_UNSUPPORTED_FEATURES*, which will ignore all but the first reply address.)

As an alternative, Anymail (for the time being) still includes a copy of the SendGrid v2 backend. See *Legacy v2 API support* below if you'd prefer to stay on the older SendGrid API.

Legacy v2 API support

Changed in version 0.8.

Anymail v0.8 switched to SendGrid's v3 Web API in its primary SendGrid email backend. SendGrid encourages all users to migrate to their v3 API.

For Anymail users who still need it, a legacy backend that calls SendGrid's earlier Web API v2 Mail Send remains available. Be aware that v2 support is considered deprecated and may be removed in a future Anymail release.

To use Anymail's SendGrid v2 backend, edit your settings.py:

```
EMAIL_BACKEND = "anymail.backends.sendgrid_v2.EmailBackend"
ANYMAIL = {
    "SENDGRID_API_KEY": "<your API key>",
}
```

The same *SENDGRID_API_KEY* will work with either Anymail's v2 or v3 SendGrid backend.

Nearly all of the documentation above for Anymail's v3 SendGrid backend also applies to the v2 backend, with the following changes:

Username/password auth (SendGrid v2 only)

SendGrid v2 allows a username/password instead of an API key (though SendGrid encourages API keys for all new installations). If you must use username/password auth, set:

```
EMAIL_BACKEND = "anymail.backends.sendgrid_v2.EmailBackend"
ANYMAIL = {
    "SENDGRID_USERNAME": "<sendgrid credential with Mail permission>",
    "SENDGRID_PASSWORD": "<password for that credential>",
    # And leave out "SENDGRID_API_KEY"
}
```

This is **not** the username/password that you use to log into SendGrid's dashboard. Create credentials specifically for sending mail in the SendGrid credentials settings.

Either username/password or *SENDGRID_API_KEY* are required (but not both).

Anymail will also look for `SENDGRID_USERNAME` and `SENDGRID_PASSWORD` at the root of the settings file if neither `ANYMAIL["SENDGRID_USERNAME"]` nor `ANYMAIL_SENDGRID_USERNAME` is set.

Duplicate attachment filenames (SendGrid v2 limitation)

Anymail is not capable of communicating multiple attachments with the same filename to the SendGrid v2 API. (This also applies to multiple attachments with *no* filename, though not to inline images.)

If you are sending multiple attachments on a single message, make sure each one has a unique, non-empty filename.

Message bodies with ESP templates (SendGrid v2 quirk)

Anymail's SendGrid v2 backend will convert empty text and HTML bodies to single spaces whenever *template_id* is set, to ensure the plaintext and HTML from your template are present in your outgoing email. This works around a limitation in SendGrid's template rendering.

Multiple Reply-To addresses (SendGrid v2 only)

Unlike SendGrid's v3 API, Anymail is able to support multiple Reply-To addresses with their v2 API.

esp_extra with SendGrid v2

Anymail's *esp_extra* attribute is merged directly with the API parameters, so the format varies between SendGrid's v2 and v3 APIs. With the v2 API, most interesting settings appear beneath `'x-smtpapi'`. Example:

```
message.esp_extra = {
    'x-smtpapi': {  # for SendGrid v2 API
        "asm_group": 1,  # Assign SendGrid unsubscribe group for this message
        "asm_groups_to_display": [1, 2, 3],
        "filters": {
            "subscriptiontrack": {  # Insert SendGrid subscription␣
↪management links
                "settings": {
                    "text/html": "If you would like to unsubscribe <% click␣
↪here %>.",
                    "text/plain": "If you would like to unsubscribe click␣
↪here: <% %>.",
                    "enable": 1
                }
            }
        }
    }
}
```

The value of `esp_extra` should be a `dict` of parameters for SendGrid's v2 mail.send API. Any keys in the dict will override Anymail's normal values for that parameter, except that `'x-smtpapi'` will be merged.

1.5.6 SparkPost

Anymail integrates with the SparkPost email service, using their python-sparkpost API client.

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.

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.)

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

1.5.7 Anymail feature support

The table below summarizes the Anymail features supported for each ESP.

Email Service Provider	*Mailgun*	*Mailjet*	*Mandrill*	*Postmark*	*SendGrid*	*SparkPost*
Anymail send options						
metadata	Yes	Yes	Yes	No	Yes	Yes
send_at	Yes	No	Yes	No	Yes	Yes
tags	Yes	Max 1 tag	Yes	Max 1 tag	Yes	Max 1 tag
track_clicks	Yes	Yes	Yes	Yes	Yes	Yes
track_opens	Yes	Yes	Yes	Yes	Yes	Yes
Batch sending/merge and ESP templates						
template_id	No	Yes	Yes	Yes	Yes	Yes
merge_data	Yes	Yes	Yes	No	Yes	Yes
merge_global_data	(emulated)	Yes	Yes	Yes	Yes	Yes
Status and event tracking						
anymail_status	Yes	Yes	Yes	Yes	Yes	Yes
AnymailTrackingEvent from webhooks	Yes	Yes	Yes	Yes	Yes	Yes
Inbound handling						
AnymailInboundEvent from webhooks	Yes	Yes	Yes	Yes	Yes	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.)

1.5.8 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')
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)
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 SSL** and a **shared authorization secret** for your Anymail webhooks. (Really, for *any* webhooks.)

Use SSL

Your Django site must use SSL, and the webhook URLs you give your ESP should 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 SSL is beyond the scope of Anymail, but there are many good tutorials on the web.

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 authorization 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 authorization 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 authorization strings. Anymail will permit webhook calls that match any of the authorization strings:

```
ANYMAIL = {
    ...
    'WEBHOOK_SECRET': [
        'abcdefghijklmnop:qrstuvwxyz0123456789',
        'ZYXWVUTSRQPONMLK:JIHGFEDCBA9876543210',
    ],
}
```

This facilitates credential rotation: first, append a new authorization string to the list, and deploy your Django site. Then, update the webhook URLs at your ESP to use the new authorization. Finally, remove the old (now unused) authorization string from the list and re-deploy.

Warning: If your webhook URLs don't use https, this shared authorization 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)

But you should start with using SSL 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 Troubleshooting

Anymail throwing errors? Not sending what you want? Here are some tips...

1.7.1 Figuring out what's wrong

Check the error message

Look for an Anymail error message in your web browser or 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 probably 90% of what you'll need to know to solve the problem.

Check your ESPs API logs

Most ESPs offer some sort of API activity log in their dashboards. Check the 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 your settings.py? (E.g., `ANYMAIL_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 won't cut it. Either specify the `from_email` explicitly on every message you send through Anymail, or add `DEFAULT_FROM_EMAIL` to your settings.py.

Try it without Anymail

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 Getting help

If you've gone through the suggestions 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! (We're not employees of any ESP.)

For questions or problems with Anymail, you can open a GitHub issue. (And if you've found a bug, you're welcome to *contribute* a fix!)

Whenever you open an issue, it's always helpful to mention which ESP you're using, include the relevant portions of your code and settings, the text of any error messages, and any exception stack traces.

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.

1.8.4 Testing

Anymail is tested on Travis against several combinations of Django and Python versions. (Full list in .travis.yml.)

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 (`pip install mock`).

To run the tests, either:

```
$ python setup.py test
```

or:

```
$ python runtests.py
```

Anymail also includes some integration tests, which do call the live ESP APIs. These integration tests require API keys (and sometimes other settings) they get from from environment variables. They're skipped if these keys aren't present. If you want to run them, look in the `*_integration_tests.py` files in the tests source for specific requirements.

1.9 Release notes

Complete release notes can be found in the project's GitHub releases page.

Anymail practices 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).

Python Module Index

a

anymail.exceptions, 25
anymail.message, 10
anymail.signals, 20

Index

A

ANYMAIL
    setting, 7
anymail.exceptions (module), 25
anymail.inbound.AnymailInboundMessage (built-in class), 28
anymail.message (module), 10
anymail.signals (module), 20
anymail.signals.AnymailInboundEvent (built-in class), 27
anymail.signals.post_send (built-in variable), 25
anymail.signals.pre_send (built-in variable), 24
ANYMAIL_IGNORE_RECIPIENT_STATUS
    setting, 7
ANYMAIL_IGNORE_UNSUPPORTED_FEATURES
    setting, 10
ANYMAIL_MAILGUN_API_KEY
    setting, 32
ANYMAIL_MAILGUN_API_URL
    setting, 33
ANYMAIL_MAILGUN_SENDER_DOMAIN
    setting, 32
ANYMAIL_MAILJET_API_KEY
    setting, 35
ANYMAIL_MAILJET_API_URL
    setting, 36
ANYMAIL_MANDRILL_API_KEY
    setting, 39
ANYMAIL_MANDRILL_API_URL
    setting, 39
ANYMAIL_MANDRILL_WEBHOOK_KEY
    setting, 39
ANYMAIL_MANDRILL_WEBHOOK_URL
    setting, 39
ANYMAIL_POSTMARK_API_URL
    setting, 44
ANYMAIL_POSTMARK_SERVER_TOKEN
    setting, 44
ANYMAIL_REQUESTS_TIMEOUT
    setting, 8
ANYMAIL_SEND_DEFAULTS
    setting, 15
ANYMAIL_SENDGRID_API_KEY
    setting, 46
ANYMAIL_SENDGRID_API_URL
    setting, 47
ANYMAIL_SENDGRID_GENERATE_MESSAGE_ID
    setting, 47
ANYMAIL_SENDGRID_MERGE_FIELD_FORMAT
    setting, 47
ANYMAIL_SENDGRID_PASSWORD
    setting, 51
ANYMAIL_SENDGRID_USERNAME
    setting, 51
ANYMAIL_SPARKPOST_API_KEY
    setting, 52
anymail_status (anymail.message.AnymailMessage attribute), 12
ANYMAIL_WEBHOOK_SECRET
    setting, 58
AnymailAPIError, 26
AnymailInboundMessage (built-in class), 30
AnymailInvalidAddress, 26
AnymailMessage (class in anymail.message), 11
AnymailMessageMixin (class in anymail.message), 16
AnymailRecipientsRefused, 26
AnymailSerializationError, 26
AnymailStatus (class in anymail.message), 13
AnymailTrackingEvent (class in anymail.signals), 20
AnymailUnsupportedFeature, 25
as_uploaded_file() (AnymailInboundMessage method), 30
attach_inline_image() (anymail.message.AnymailMessage method), 12
attach_inline_image() (in module anymail.message), 15
attach_inline_image_file() (anymail.message.AnymailMessage method), 12

attach_inline_image_file() (in module anymail.message), 14
attachments (anymail.inbound.AnymailInboundMessage attribute), 29

C

cc (anymail.inbound.AnymailInboundMessage attribute), 29
click_url (anymail.signals.AnymailTrackingEvent attribute), 22

D

date (anymail.inbound.AnymailInboundMessage attribute), 29
description (anymail.signals.AnymailTrackingEvent attribute), 22

E

envelope_recipient (anymail.inbound.AnymailInboundMessage attribute), 28
envelope_sender (anymail.inbound.AnymailInboundMessage attribute), 28
esp_event (anymail.signals.AnymailInboundEvent attribute), 28
esp_event (anymail.signals.AnymailTrackingEvent attribute), 22
esp_extra (anymail.message.AnymailMessage attribute), 12
esp_response (anymail.message.AnymailStatus attribute), 14
event_id (anymail.signals.AnymailInboundEvent attribute), 28
event_id (anymail.signals.AnymailTrackingEvent attribute), 21
event_type (anymail.signals.AnymailInboundEvent attribute), 27
event_type (anymail.signals.AnymailTrackingEvent attribute), 20

F

from_email (anymail.inbound.AnymailInboundMessage attribute), 28

G

get_content_bytes() (AnymailInboundMessage method), 31
get_content_disposition() (AnymailInboundMessage method), 31
get_content_maintype() (AnymailInboundMessage method), 30
get_content_subtype() (AnymailInboundMessage method), 30
get_content_text() (AnymailInboundMessage method), 31
get_content_type() (AnymailInboundMessage method), 30
get_filename() (AnymailInboundMessage method), 30

H

html (anymail.inbound.AnymailInboundMessage attribute), 29

I

inline_attachments (anymail.inbound.AnymailInboundMessage attribute), 29
is_attachment() (AnymailInboundMessage method), 30
is_inline_attachment() (AnymailInboundMessage method), 31

M

merge_data (anymail.message.AnymailMessage attribute), 18
merge_global_data (anymail.message.AnymailMessage attribute), 18
message (anymail.signals.AnymailInboundEvent attribute), 27
message_id (anymail.message.AnymailStatus attribute), 13
message_id (anymail.signals.AnymailTrackingEvent attribute), 21
metadata (anymail.message.AnymailMessage attribute), 11
metadata (anymail.signals.AnymailTrackingEvent attribute), 22
mta_response (anymail.signals.AnymailTrackingEvent attribute), 22

P

Python Enhancement Proposals
    PEP 8, 62

R

recipient (anymail.signals.AnymailTrackingEvent attribute), 21
recipients (anymail.message.AnymailStatus attribute), 14
reject_reason (anymail.signals.AnymailTrackingEvent attribute), 22
RFC
    RFC 2822, 13
    RFC 5322, 26

S

send_at (anymail.message.AnymailMessage attribute), 12
setting

ANYMAIL, 7
ANYMAIL_IGNORE_RECIPIENT_STATUS, 7
ANYMAIL_IGNORE_UNSUPPORTED_FEATURES, 10
ANYMAIL_MAILGUN_API_KEY, 32
ANYMAIL_MAILGUN_API_URL, 33
ANYMAIL_MAILGUN_SENDER_DOMAIN, 32
ANYMAIL_MAILJET_API_KEY, 35
ANYMAIL_MAILJET_API_URL, 36
ANYMAIL_MANDRILL_API_KEY, 39
ANYMAIL_MANDRILL_API_URL, 39
ANYMAIL_MANDRILL_WEBHOOK_KEY, 39
ANYMAIL_MANDRILL_WEBHOOK_URL, 39
ANYMAIL_POSTMARK_API_URL, 44
ANYMAIL_POSTMARK_SERVER_TOKEN, 44
ANYMAIL_REQUESTS_TIMEOUT, 8
ANYMAIL_SEND_DEFAULTS, 15
ANYMAIL_SENDGRID_API_KEY, 46
ANYMAIL_SENDGRID_API_URL, 47
ANYMAIL_SENDGRID_GENERATE_MESSAGE_ID, 47
ANYMAIL_SENDGRID_MERGE_FIELD_FORMAT, 47
ANYMAIL_SENDGRID_PASSWORD, 51
ANYMAIL_SENDGRID_USERNAME, 51
ANYMAIL_SPARKPOST_API_KEY, 52
ANYMAIL_WEBHOOK_SECRET, 58
spam_detected (anymail.inbound.AnymailInboundMessage attribute), 29
spam_score (anymail.inbound.AnymailInboundMessage attribute), 29
status (anymail.message.AnymailStatus attribute), 13
stripped_html (anymail.inbound.AnymailInboundMessage attribute), 29
stripped_text (anymail.inbound.AnymailInboundMessage attribute), 29
subject (anymail.inbound.AnymailInboundMessage attribute), 29

T

tags (anymail.message.AnymailMessage attribute), 11
tags (anymail.signals.AnymailTrackingEvent attribute), 22
template_id (anymail.message.AnymailMessage attribute), 17
text (anymail.inbound.AnymailInboundMessage attribute), 29
timestamp (anymail.signals.AnymailInboundEvent attribute), 27
timestamp (anymail.signals.AnymailTrackingEvent attribute), 21
to (anymail.inbound.AnymailInboundMessage attribute), 28
track_clicks (anymail.message.AnymailMessage attribute), 12
track_opens (anymail.message.AnymailMessage attribute), 11

U

user_agent (anymail.signals.AnymailTrackingEvent attribute), 22