Sending email

Although Python provides a mail sending interface via the `smtplib` module, Django provides a couple of light wrappers over it. These wrappers are provided to make sending email extra quick, to help test email sending during development, and to provide support for platforms that can't use SMTP.

The code lives in the `django.core.mail` module.

Quick example

In two lines:

```
from django.core.mail import send_mail

send_mail(
    "Subject here",
    "Here is the message.",
    "from@example.com",
    ["to@example.com"],
    fail_silently=False,
)
```

Mail is sent using the SMTP host and port specified in the `EMAIL_HOST` and `EMAIL_PORT` settings. The `EMAIL_HOST_USER` and `EMAIL_HOST_PASSWORD` settings, if set, are used to authenticate to the SMTP server, and the `EMAIL_USE_TLS` and `EMAIL_USE_SSL` settings control whether a secure connection is used.

Note: The character set of email sent with `django.core.mail` will be set to the value of your `DEFAULT_CHARSET` setting.

`send_mail()`

`send_mail(subject, message, from_email, recipient_list, fail_silently=False, auth_user=None, auth_password=None, connection=None, html_message=None)` [source]

In most cases, you can send email using `django.core.mail.send_mail()`.

The `subject`, `message`, `from_email` and `recipient_list` parameters are required.

- `subject`: A string.
- `message`: A string.
- `from_email`: A string. If `None`, Django will use the value of the `DEFAULT_FROM_EMAIL` setting.
- `recipient_list`: A list of strings, each an email address. Each member of `recipient_list` will see the other recipients in the "To:" field of the email message.
- `fail_silently`: A boolean. When it's `False`, `send_mail()` will raise an `smtplib.SMTPException` if an error occurs. See the `smtplib` docs for a list of possible exceptions, all of which are subclasses of `SMTPException`.
- `auth_user`: The optional username to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the `EMAIL_HOST_USER` setting.
- `auth_password`: The optional password to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the `EMAIL_HOST_PASSWORD` setting.
- `connection`: The optional email backend to use to send the mail. If unspecified, an instance of the default backend will be used. See the documentation on Email backends for more details.
- `html_message`: If `html_message` is provided, the resulting email will be a *multipart/alternative* email with `message` as the *text/plain* content type and `html_message` as the *text/html* content type.

The return value will be the number of successfully delivered messages (which can be `0` or `1` since it can only send one message).

`send_mass_mail()`

`send_mass_mail(datatuple, fail_silently=False, auth_user=None, auth_password=None, connection=None)` [source]

`django.core.mail.send_mass_mail()` is intended to handle mass emailing.

`datatuple` is a tuple in which each element is in this format:

```
(subject, message, from_email, recipient_list)
```

`fail_silently`, `auth_user` and `auth_password` have the same functions as in `send_mail()`.

Each separate element of `datatuple` results in a separate email message. As in `send_mail()`, recipients in the same `recipient_list` will all see the other addresses in the email messages' "To:" field.

For example, the following code would send two different messages to two different sets of recipients; however, only one connection to the mail server would be opened:

```
message1 = (
    "Subject here",
    "Here is the message",
    "from@example.com",
    ["first@example.com", "other@example.com"],
)
message2 = (
    "Another Subject",
    "Here is another message",
    "from@example.com",
    ["second@test.com"],
)
send_mass_mail((message1, message2), fail_silently=False)
```

The return value will be the number of successfully delivered messages.

`send_mass_mail()` vs. `send_mail()`

The main difference between `send_mass_mail()` and `send_mail()` is that `send_mail()` opens a connection to the mail server each time it's executed, while `send_mass_mail()` uses a single connection for all of its messages. This makes `send_mass_mail()` slightly more efficient.

`mail_admins()`

`mail_admins(subject, message, fail_silently=False, connection=None, html_message=None)` [source]

`django.core.mail.mail_admins()` is a shortcut for sending an email to the site admins, as defined in the `ADMINS` setting.

`mail_admins()` prefixes the subject with the value of the `EMAIL_SUBJECT_PREFIX` setting, which is `"[Django] "` by default.

The "From:" header of the email will be the value of the `SERVER_EMAIL` setting.

This method exists for convenience and readability.

If `html_message` is provided, the resulting email will be a *multipart/alternative* email with `message` as the *text/plain* content type and `html_message` as the *text/html* content type.

`mail_managers()`

`mail_managers(subject, message, fail_silently=False, connection=None, html_message=None)` [source]

`django.core.mail.mail_managers()` is just like `mail_admins()`, except it sends an email to the site managers, as defined in the `MANAGERS` setting.

Examples

This sends a single email to john@example.com and jane@example.com, with them both appearing in the "To:":

```
send_mail(
    "Subject",
    "Message.",
    "from@example.com",
    ["john@example.com", "jane@example.com"],
)
```

This sends a message to john@example.com and jane@example.com, with them both receiving a separate email:

```
datatuple = (
    ("Subject", "Message.", "from@example.com", ["john@example.com"]),
    ("Subject", "Message.", "from@example.com", ["jane@example.com"]),
)
send_mass_mail(datatuple)
```

Preventing header injection

Header injection is a security exploit in which an attacker inserts extra email headers to control the "To:" and "From:" in email messages that your scripts generate.

The Django email functions outlined above all protect against header injection by forbidding newlines in header values. If any `subject`, `from_email` or `recipient_list` contains a newline (in either Unix, Windows or Mac style), the email function (e.g. `send_mail()`) will raise `django.core.mail.BadHeaderError` (a subclass of `ValueError`) and, hence, will not send the email. It's your responsibility to validate all data before passing it to the email functions.

If a `message` contains headers at the start of the string, the headers will be printed as the first bit of the email message.

Here's an example view that takes a `subject`, `message` and `from_email` from the request's POST data, sends that to admin@example.com and redirects to "/contact/thanks/" when it's done:

```
from django.core.mail import BadHeaderError, send_mail
from django.http import HttpResponse, HttpResponseRedirect


def send_email(request):
    subject = request.POST.get("subject", "")
    message = request.POST.get("message", "")
    from_email = request.POST.get("from_email", "")
    if subject and message and from_email:
        try:
            send_mail(subject, message, from_email, ["admin@example.com"])
        except BadHeaderError:
            return HttpResponse("Invalid header found.")
        return HttpResponseRedirect("/contact/thanks/")
    else:
        # In reality we'd use a form class
        # to get proper validation errors.
        return HttpResponse("Make sure all fields are entered and valid.")
```

The `EmailMessage` class

Django's `send_mail()` and `send_mass_mail()` functions are actually thin wrappers that make use of the `EmailMessage` class.

Not all features of the `EmailMessage` class are available through the `send_mail()` and related wrapper functions. If you wish to use advanced features, such as BCC'ed recipients, file attachments, or multi-part email, you'll need to create `EmailMessage` instances directly.

Note: This is a design feature. `send_mail()` and related functions were originally the only interface Django provided. However, the list of parameters they accepted was slowly growing over time. It made sense to move to a more object-oriented design for email messages and retain the original functions only for backwards compatibility.

`EmailMessage` is responsible for creating the email message itself. The email backend is then responsible for sending the email.

For convenience, `EmailMessage` provides a `send()` method for sending a single email. If you need to send multiple messages, the email backend API provides an alternative.

`EmailMessage` Objects

`class EmailMessage` [source]

The `EmailMessage` class is initialized with the following parameters (in the given order, if positional arguments are used). All parameters are optional and can be set at any time prior to calling the `send()` method.

- `subject`: The subject line of the email.
- `body`: The body text. This should be a plain text message.
- `from_email`: The sender's address. Both `fred@example.com` and `"Fred" <fred@example.com>` forms are legal. If omitted, the `DEFAULT_FROM_EMAIL` setting is used.
- `to`: A list or tuple of recipient addresses.
- `bcc`: A list or tuple of addresses used in the "Bcc" header when sending the email.
- `connection`: An email backend instance. Use this parameter if you want to use the same connection for multiple messages. If omitted, a new connection is created when `send()` is called.
- `attachments`: A list of attachments to put on the message. These can be either `MIMEBase` instances, or `(filename, content, mimetype)` triples.
- `headers`: A dictionary of extra headers to put on the message. The keys are the header name, values are the header values. It's up to the caller to ensure header names and values are in the correct format for an email message. The corresponding attribute is `extra_headers`.
- `cc`: A list or tuple of recipient addresses used in the "Cc" header when sending the email.

- `reply_to` : A list or tuple of recipient addresses used in the "Reply-To" header when sending the email.

For example:

```
from django.core.mail import EmailMessage

email = EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to1@example.com", "to2@example.com"],
    ["bcc@example.com"],
    reply_to=["another@example.com"],
    headers={"Message-ID": "foo"},
)
```

The class has the following methods:

- `send(fail_silently=False)` sends the message. If a connection was specified when the email was constructed, that connection will be used. Otherwise, an instance of the default backend will be instantiated and used. If the keyword argument `fail_silently` is `True` , exceptions raised while sending the message will be quashed. An empty list of recipients will not raise an exception. It will return `1` if the message was sent successfully, otherwise `0` .
- `message()` constructs a `django.core.mail.SafeMIMEText` object (a subclass of Python's `MIMEText` class) or a `django.core.mail.SafeMIMEMultipart` object holding the message to be sent. If you ever need to extend the `EmailMessage` class, you'll probably want to override this method to put the content you want into the MIME object.
- `recipients()` returns a list of all the recipients of the message, whether they're recorded in the `to` , `cc` or `bcc` attributes. This is another method you might need to override when subclassing, because the SMTP server needs to be told the full list of recipients when the message is sent. If you add another way to specify recipients in your class, they need to be returned from this method as well.
- `attach()` creates a new file attachment and adds it to the message. There are two ways to call `attach()` :
  - You can pass it a single argument that is a `MIMEBase` instance. This will be inserted directly into the resulting message.
  - Alternatively, you can pass `attach()` three arguments: `filename` , `content` and `mimetype` . `filename` is the name of the file attachment as it will appear in the email, `content` is the data that will be contained inside the attachment and `mimetype` is the optional MIME type for the attachment. If you omit `mimetype` , the MIME content type will be guessed from the filename of the attachment.

    For example:

    ```
    message.attach("design.png", img_data, "image/png")
    ```

    If you specify a `mimetype` of *message/rfc822*, it will also accept `django.core.mail.EmailMessage` and `email.message.Message`.

    For a `mimetype` starting with *text/*, content is expected to be a string. Binary data will be decoded using UTF-8, and if that fails, the MIME type will be changed to *application/octet-stream* and the data will be attached unchanged.

    In addition, *message/rfc822* attachments will no longer be base64-encoded in violation of **RFC 2046#section-5.2.1**, which can cause issues with displaying the attachments in Evolution and Thunderbird.
- `attach_file()` creates a new attachment using a file from your filesystem. Call it with the path of the file to attach and, optionally, the MIME type to use for the attachment. If the MIME type is omitted, it will be guessed from the filename. You can use it like this:

  ```
  message.attach_file("/images/weather_map.png")
  ```

  For MIME types starting with *text/*, binary data is handled as in `attach()` .

Sending alternative content types

Sending multiple content versions

It can be useful to include multiple versions of the content in an email; the classic example is to send both text and HTML versions of a message. With Django's email library, you can do this using the `EmailMultiAlternatives` class.

```
class EmailMultiAlternatives                                                    [source]
```

A subclass of `EmailMessage` that has an additional `attach_alternative()` method for including extra versions of the message body in the email. All the other methods (including the class initialization) are inherited directly from `EmailMessage`.

```
attach_alternative(content, mimetype)                                           [source]
```

Attach an alternative representation of the message body in the email.

For example, to send a text and HTML combination, you could write:

```
from django.core.mail import EmailMultiAlternatives

subject = "hello"
from_email = "from@example.com"
to = "to@example.com"
text_content = "This is an important message."
html_content = "<p>This is an <strong>important</strong> message.</p>"
msg = EmailMultiAlternatives(subject, text_content, from_email, [to])
msg.attach_alternative(html_content, "text/html")
msg.send()
```

Updating the default content type

By default, the MIME type of the `body` parameter in an `EmailMessage` is `"text/plain"`. It is good practice to leave this alone, because it guarantees that any recipient will be able to read the email, regardless of their mail client. However, if you are confident that your recipients can handle an alternative content type, you can use the `content_subtype` attribute on the `EmailMessage` class to change the main content type. The major type will always be `"text"`, but you can change the subtype. For example:

```
msg = EmailMessage(subject, html_content, from_email, [to])
msg.content_subtype = "html"  # Main content is now text/html
msg.send()
```

Email backends

The actual sending of an email is handled by the email backend.

The email backend class has the following methods:

- `open()` instantiates a long-lived email-sending connection.
- `close()` closes the current email-sending connection.
- `send_messages(email_messages)` sends a list of `EmailMessage` objects. If the connection is not open, this call will implicitly open the connection, and close the connection afterward. If the connection is already open, it will be left open after mail has been sent.

It can also be used as a context manager, which will automatically call `open()` and `close()` as needed:

```
from django.core import mail

with mail.get_connection() as connection:
    mail.EmailMessage(
        subject1,
        body1,
        from1,
        [to1],
        connection=connection,
    ).send()
    mail.EmailMessage(
        subject2,
        body2,
        from2,
        [to2],
        connection=connection,
    ).send()
```

Obtaining an instance of an email backend

The `get_connection()` function in `django.core.mail` returns an instance of the email backend that you can use.

```
get_connection(backend=None, fail_silently=False, *args, **kwargs)
```
[source]

By default, a call to `get_connection()` will return an instance of the email backend specified in `EMAIL_BACKEND`. If you specify the `backend` argument, an instance of that backend will be instantiated.

The `fail_silently` argument controls how the backend should handle errors. If `fail_silently` is True, exceptions during the email sending process will be silently ignored.

All other arguments are passed directly to the constructor of the email backend.

Django ships with several email sending backends. With the exception of the SMTP backend (which is the default), these backends are only useful during testing and development. If you have special email sending requirements, you can write your own email backend.

SMTP backend

```
class backends.smtp.EmailBackend(host=None, port=None, username=None, password=None, use_tls=None, fail_silently=False, use_ssl=None, timeout=None, ssl_keyfile=None, ssl_certfile=None, **kwargs)
```

This is the default backend. Email will be sent through a SMTP server.

The value for each argument is retrieved from the matching setting if the argument is `None`:

- `host`: `EMAIL_HOST`
- `port`: `EMAIL_PORT`
- `username`: `EMAIL_HOST_USER`
- `password`: `EMAIL_HOST_PASSWORD`
- `use_tls`: `EMAIL_USE_TLS`
- `use_ssl`: `EMAIL_USE_SSL`
- `timeout`: `EMAIL_TIMEOUT`
- `ssl_keyfile`: `EMAIL_SSL_KEYFILE`
- `ssl_certfile`: `EMAIL_SSL_CERTFILE`

The SMTP backend is the default configuration inherited by Django. If you want to specify it explicitly, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.smtp.EmailBackend"
```

If unspecified, the default `timeout` will be the one provided by `socket.getdefaulttimeout()`, which defaults to `None` (no timeout).

Console backend

Instead of sending out real emails the console backend just writes the emails that would be sent to the standard output. By default, the console backend writes to `stdout`. You can use a different stream-like object by providing the `stream` keyword argument when constructing the connection.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

File backend

The file backend writes emails to a file. A new file is created for each new session that is opened on this backend. The directory to which the files are written is either taken from the `EMAIL_FILE_PATH` setting or from the `file_path` keyword when creating a connection with `get_connection()`.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.filebased.EmailBackend"
EMAIL_FILE_PATH = "/tmp/app-messages"  # change this to a proper location
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

In-memory backend

The `'locmem'` backend stores messages in a special attribute of the `django.core.mail` module. The `outbox` attribute is created when the first message is sent. It's a list with an `EmailMessage` instance for each message that would be sent.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.locmem.EmailBackend"
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development and testing.

Django's test runner automatically uses this backend for testing.

Dummy backend

As the name suggests the dummy backend does nothing with your messages. To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = "django.core.mail.backends.dummy.EmailBackend"
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

Defining a custom email backend

If you need to change how emails are sent you can write your own email backend. The `EMAIL_BACKEND` setting in your settings file is then the Python import path for your backend class.

Custom email backends should subclass `BaseEmailBackend` that is located in the `django.core.mail.backends.base` module. A custom email backend must implement the `send_messages(email_messages)` method. This method receives a list of `EmailMessage` instances and returns the number of successfully delivered messages. If your backend has any concept of a persistent session or connection, you should also implement the `open()` and `close()` methods. Refer to `smtp.EmailBackend` for a reference implementation.

Sending multiple emails

Establishing and closing an SMTP connection (or any other network connection, for that matter) is an expensive process. If you have a lot of emails to send, it makes sense to reuse an SMTP connection, rather than creating and destroying a connection every time you want to send an email.

There are two ways you tell an email backend to reuse a connection.

Firstly, you can use the `send_messages()` method. `send_messages()` takes a list of `EmailMessage` instances (or subclasses), and sends them all using a single connection.

For example, if you have a function called `get_notification_email()` that returns a list of `EmailMessage` objects representing some periodic email you wish to send out, you could send these emails using a single call to send_messages:

```
from django.core import mail

connection = mail.get_connection()  # Use default email connection
messages = get_notification_email()
connection.send_messages(messages)
```

In this example, the call to `send_messages()` opens a connection on the backend, sends the list of messages, and then closes the connection again.

The second approach is to use the `open()` and `close()` methods on the email backend to manually control the connection. `send_messages()` will not manually open or close the connection if it is already open, so if you manually open the connection, you can control when it is closed. For example:

```
from django.core import mail

connection = mail.get_connection()

# Manually open the connection
connection.open()

# Construct an email message that uses the connection
email1 = mail.EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to1@example.com"],
    connection=connection,
)
email1.send()  # Send the email

# Construct two more messages
email2 = mail.EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to2@example.com"],
)
email3 = mail.EmailMessage(
    "Hello",
    "Body goes here",
    "from@example.com",
    ["to3@example.com"],
)

# Send the two emails in a single call -
connection.send_messages([email2, email3])
# The connection was already open so send_messages() doesn't close it.
# We need to manually close the connection.
connection.close()
```

Configuring email for development

There are times when you do not want Django to send emails at all. For example, while developing a website, you probably don't want to send out thousands of emails – but you may want to validate that emails will be sent to the right people under the right conditions, and that those emails will contain the correct content.

The easiest way to configure email for local development is to use the console email backend. This backend redirects all email to `stdout`, allowing you to inspect the content of mail.

The file email backend can also be useful during development – this backend dumps the contents of every SMTP connection to a file that can be inspected at your leisure.

Another approach is to use a "dumb" SMTP server that receives the emails locally and displays them to the terminal, but does not actually send anything. The aiosmtpd package provides a way to accomplish this:

```
python -m pip install aiosmtpd

python -m aiosmtpd -n -l localhost:8025
```

This command will start a minimal SMTP server listening on port 8025 of localhost. This server prints to standard output all email headers and the email body. You then only need to set the `EMAIL_HOST` and `EMAIL_PORT` accordingly. For a more detailed discussion of SMTP server options, see the documentation of the aiosmtpd module.

For information about unit-testing the sending of emails in your application, see the Email services section of the testing documentation.