Contents

1 Introduction 3
  1.1 Installation .................................................. 3

2 Quick Start 5

3 Class-based emails 7
  3.1 Basic Example ............................................... 7
  3.2 Emails with dynamic values .................................. 8
  3.3 HTML Emails .................................................. 8
  3.4 Call Signature ................................................ 9
  3.5 About as_callable(**kwargs) ............................... 10

4 emailtools.cbe.base 11
  4.1 BaseEmail .................................................... 11
  4.2 BasicEmail ................................................... 12
  4.3 HTMLEmail ................................................... 13
  4.4 MarkdownEmail ............................................... 13

5 How To 15
  5.1 Sending a Welcome Email ................................... 15
  5.2 Password Reset Email ....................................... 15

6 Changelog 21
  6.1 0.2.2 (2014-07-04) ........................................... 21
  6.2 0.2.1 (2013-10-30) ........................................... 21
  6.3 0.2.0 (2013-09-12) ........................................... 21
  6.4 0.1.1 (2013-07-27) ........................................... 21

7 Development 23
Tools to assist in sending emails from your Django app.

Contents:
django-emailtools takes an approach to sending emails similar to the class-based view’s approach to view callables. At Fusionbox we’ve found that our email sending often follows a predictable pattern and class-based emails arose from that pattern.

Ultimately, the goal of class-based emails is to

Installation

1. Install the package:

   $ pip install django-emailtools

   Or you can install it from source:

   $ pip install -e git://github.com/fusionbox/django-emailtools@master#egg=django-emailtools-dev

2. Add emailtools to your INSTALLED_APPS.
Write your email class and instantiate your email callable.

```python
# app/emails.py
from emailtools import BasicEmail
class WelcomeEmail(BasicEmail):
    from_email = 'admin@example.com'
    subject = 'Welcome to example.com'
    template_name = 'welcome_email.html'

    def get_to(self):
        return [self.args[0]]

    def get_context_data(self, **kwargs):
        kwargs = super(WelcomeEmail, self).get_context_data(**kwargs)
        kwargs['email_address'] = self.args[0]
        return kwargs

send_welcome_email = WelcomeEmail.as_callable()
```

Write your email template.

```html
# app/templates/welcome_email.html
<html>
<body>
    <h1>Welcome to example.com</h1>
    <p>Here is some content about how happy we are to have you at example.com. You can now login using '{{ email_address }}' as your username</p>
</body>
</html>
```

And to send the email.
>>> from app.emails import send_welcome_email
>>> send_welcome_email('joe.smith@example.com')  # Sends the email

Or if you wanted to do something with the message before you sent it:

```python
>>> from myapp.emails import WelcomeEmail
>>> email_instance = WelcomeEmail
>>> message = email_instance.get_email_message()
>>> message
<django.core.mail.message.EmailMultiAlternatives at 0x10668d150>
>>> message.send()  # Sends the email.
```
Class-based emails

django-emailtools takes an approach to sending emails similar to the class-based view’s approach to view callables. At Fusionbox we’ve found that our email sending often follows a predictable pattern and class-based emails arose from that pattern.

Ultimately, the goal of class-based emails is to assist developers in following the DRY principle and reuse code through inheritance and mixin classes.

**Basic Example**

A very basic example of sending emails in django using the built in `send_mail` function might look something like the following.

```python
from django.core.mail import send_mail

def send_registration_email():
    send_mail(
        'A new user has registered on example.com.',
        'A user has registered',
        'admin@example.com',
        ['webmaster@example.com'],
    )
```

Now, here is the same example using class based emails.

```python
from emailtools.cbe import BasicEmail

class RegisteredEmail(BasicEmail):
    to = 'webmaster@example.com'
    from_email = 'admin@example.com'
    subject = 'A user has registered'
    body = 'A new user has registered on example.com.'

send_registration_email = UserRegisteredEmail.as_callable()
```
In both examples, calling the `send_registration_email` function will send an email to `webmaster@example.com` from the address `webmaster@example.com` with the subject "A user has registered" and with the message body "A new user has registered on example.com". Admittedly, this example is not very useful, so let's look at making some of these values more dynamic.

**Emails with dynamic values**

Now, let's write another example, in which our message body and the email recipient list and message body are dynamic.

```python
# accounts/emails.py
from emailtools.cbe import BasicEmail

class WelcomeEmail(BasicEmail):
    from_email = 'admin@example.com'
    subject = 'Welcome to example.com'
    
    def get_to(self):
        return [self.args[0].email]
    
    def get_body(self):
        return 'Dear {user.username},
Welcome to example.com,
- The example.com Team'.format(user=self.args[0])

send_welcome_email = WelcomeEmail.as_callable()
```

Our new `send_welcome_email` function expects a single argument which it expects to be a `user` instance, from which it will extract the `username` for the message body, and the `to` address. To send our email, we just call the `send_welcome_email` function with a user instance.

```python
>>> from app.emails import send_welcome_email
>>> user = User.objects.get(...)  
>>> send_welcome_email(user)  # Sends the welcome email.
```

**Note:** The `BasicEmail` class is essentially a wrapper around the `django.core.email.EmailMessage` class with both properties and method hooks for configuring, instantiating, and sending emails using that class.

**HTML Emails**

While the simple examples above may work well for simple emails, most modern web applications are not just sending plain text emails. `emailtools` ships with two solutions for constructing and sending emails with both a plain text message and an html message. Both the `HTMLEmail` and `MarkdownEmail` classes extend `django.core.email.EmailMultiAlternative`, and uses django's built in template engine to set the html message on the email.

Let's rewrite the welcome email class to send an html message.
from emailtools import HTMLEmail

class WelcomeEmail(HTMLEmail):
    template_name = 'app/welcome_email.html'
    from_email = 'admin@example.com'
    subject = 'Welcome to example.com'

    def get_to(self):
        return [self.args[0].email]

    def get_context_data(self, **kwargs):
        kwargs = super(WelcomeEmail, self).get_context_data(**kwargs)
        kwargs['user'] = self.args[0]
        return kwargs

send_welcome_email = WelcomeEmail.as_callable()

And now our template.

# app/templates/app/welcome_email.html
<h1>Welcome to example.com</h1>
<p>Dear {{ user.email }}</p>
<p>Thank you for signing up to <a href="http://www.example.com">example.com</a></p>
<p>The example.com team</p>

Now, our message will be rendered using the template engine.

## Call Signature

Up until now, accessing the calling arguments for our email function has involved accessing them in `self.args` or `self.kwargs`, which is both ugly and unintuitive. If you take a look at the `__init__` method of `BaseEmail` you’ll see that it merely sets `*args` and `**kwargs as self.args` and `self.kwargs`. This is the default behavior for all email classes, and it is entirely in the developers hands to override this in any way you please.

Here is a slightly modified version of our `WelcomeEmail` that demonstrates this concept.

from emailtools import HTMLEmail

class WelcomeEmail(HTMLEmail):
    template_name = 'app/welcome_email.html'
    from_email = 'admin@example.com'
    subject = 'Welcome to example.com'

    def __init__(self, user):
        self.user = user
        self.to = [user.email]

    def get_context_data(self, **kwargs):
        kwargs = super(WelcomeEmail, self).get_context_data(**kwargs)
        kwargs['user'] = self.args[0]
        return kwargs

send_welcome_email = WelcomeEmail.as_callable()

We gain readability, and validation that the caller complied with the call signature of our email class. In this example, we didn’t call `super` on `__init__`, which is fine. The `__init__` method is yours to override and modify in whatever
way suites the needs of your application.

**About as_callable(**kwargs)**

At this point, if you’ve used class based views, you should be noticing some similarities in `as_callable` and `as_view.as_callable` returns a callable function that will send the email. By default, any `*args` and `**kwargs` passed into the email callable are accessible via `self.args` and `self.kwargs`, similar to class based views. This however is only the default implementation of the `__init__` method for class based emails. You may override the `__init__` method however you would like.

From our example above, the following two ways of sending emails are effectively the same.

```python
>>> from my_app.emails import WelcomeEmail
>>> send_welcome_email = WelcomeEmail.as_callable()
>>> send_welcome_email(user)  # Sends the email.
>>> email_instance = WelcomeEmail(user)
>>> email_instance.send()
```

Directly calling the email callable, and calling `send()` on the instantiated email class are identical.
This document provides API reference material for the classes and mixins found in `emailtools`.

**BaseEmail**

```python
class emailtools.cbe.base.BaseEmail:
    This is the base class for all class based emails. While not usable on its own, it establishes the base api for email sending.

    email_message_class
        The class that will be used to construct the email message.

    get_email_message_class()
        Returns the email message class.

    get_email_message_kwargs()
        Constructs and returns the kwargs that will be used to instantiate the email message.

    get_email_message()
        Constructs and returns the instantiated email message.

    get_send_kwargs(**kwargs)
        Constructs and returns the kwargs that will be passed to the send method of the instantiated email message.

    send()
        Constructs and sends the email message.

    as_callable()
        Returns the email callable that can be used to send the email message, or construct and return the unsent email message.
```
BasicEmail

class emailtools.cbe.base.BasicEmail

This class is the simplest implementation for class-based emails, and can be thought of as a loose wrapper around django.core.email.EmailMessage. The class provides properties and methods for declaring or building each of the arguments used to instantiate and send an EmailMessage.

```python
e-mail_message_class
  *default: django.core.email.EmailMessage

``subject``
  Static property to be used as the subject attribute for the email message.

``to``
  Static property to be used as the to address for the email message.

``cc``
  Static property to be used as the cc attribute for the email message.

``bcc``
  Static property to be used as the bcc attribute for the email message.

``from_email``
  Static property to be used as the from_email attribute for the email message.

``body``
  Static property to be used as the body attribute for the email message.

``connection``
  Static property to be used as the connection to be used for sending the email message.

``attachments``
  Static property to be used for the attachments of the email message.

``headers``
  Static property to be used for the headers of the email message.

``fail_silently``
  Passed to the send method of the email message, to determine whether exceptions raised while sending should be squashed.

``get_to``
  Returns the list of email addresses the email addresses should be sent to.

``get_cc``
  Returns the list of email addresses the email addresses should be cc’d.

``get_bcc``
  Returns the list of email addresses the email addresses should be bcc’d.

``get_from_email``
  Returns the email address the email addresses will be from.

``get_body``
  Returns the message body for the email message.

``get_connection``
  Returns the email connection to be used for sending the email message.

``get_headers``
  Returns any headers to be added to the email message.
```
### HTMLEmail

```python
class emailtools.cbe.base.HTMLEmail
    This class leverages the django template engine to attach an html message to the email message.

    email_message_class
        *default: django.core.email.EmailMessage
    template_name
        Path to the template that should be used for rendering the body of the message.

    get_context_data(**kwargs)
        Constructs and returns the context to be used for template rendering.
```

### MarkdownEmail

```python
class emailtools.cbe.base.MarkdownEmail
    Similar to HTMLEmail, this class uses the django template engine for rendering an html email message. It however expects a template written in markdown, which is then inserted into the body of a base html template.

    layout_template
        Declares what template should be used as the base layout for this email. This template should expect a context variable content which will contain the rendered markdown of the message.

    template_name
        Path to the template that should be used for rendering the body of the message. This template is rendered as markdown and then inserted into the template returned by get_layout_template().

    get_context_data(**kwargs)
        Constructs and returns the context to be used for template rendering.

    get_layout_template()
        Returns the template that should be used for rendering the base html layout of the message. If layout_template is not set on the class, this method returns the value found in settings.EMAIL_LAYOUT. This template will be rendered with the context returned by get_layout_context_data(), which by default contains one value content which is the rendered markdown content.

    get_layout_context_data(**kwargs)
        Constructs and renders the context data for the base layout template. By default, this returns a context with a single value content which contains the rendered markdown content from the markdown template.
```
How To

Here we will explore some examples using class-based emails.

Sending a Welcome Email

In the introduction to class-based emails, we demonstrated some very basic patterns for sending a welcome email. Here, we will expand on those patterns a bit further to demonstrate a real use case example.

Our goal here will be to send a welcome email to newly registered users by hooking into the \texttt{post\_save} signal provided by Django.

Password Reset Email

Sending a password reset email manually. First let's take a look at how Django does this in the built-in \texttt{PasswordResetForm} packaged with \texttt{django.contrib.auth}:

```python
class PasswordResetForm(forms.Form):
    email = forms.EmailField(label=_("Email"), max_length=254)

    def save(self, domain_override=None,
             subject_template_name='registration/password_reset_subject.txt',
             email_template_name='registration/password_reset_email.html',
             use_https=False, token_generator=default_token_generator,
             from_email=None, request=None):
        
        """
        Generates a one-use only link for resetting password and sends to the
        user.
        """
        from django.core.mail import send_mail
        UserModel = get_user_model()
        email = self.cleaned_data["email"]
        users = UserModel._default_manager.filter(email__iexact=email)
        for user in users:
            context = {
                'use_https': use_https,
                'token': token_generator.make_token(user),
                'protocol': 'https' if use_https else 'http',
                'domain': domain_override or get_current_site(request).domain,
                'site_name': get_current_site(request).name,
            }
            send_mail(subject_template_name, email_template_name, context, [user.email],
                      fail_silently=True, from_email=from_email)
        
        return urlconf_namespace + urlconf_namespace_url
```

for user in users:
    # Make sure that no email is sent to a user that actually has
    # a password marked as unusable
    if user.password == UNUSABLE_PASSWORD:
        continue
    if not domain_override:
        current_site = get_current_site(request)
        site_name = current_site.name
        domain = current_site.domain
    else:
        site_name = domain = domain_override
    c = {
        'email': user.email,
        'domain': domain,
        'site_name': site_name,
        'uid': int_to_base36(user.pk),
        'user': user,
        'token': token_generator.make_token(user),
        'protocol': 'https' if use_https else 'http',
        }
    subject = loader.render_to_string(subject_template_name, c)
    # Email subject *must not* contain newlines
    subject = ''.join(subject.splitlines())
    email = loader.render_to_string(email_template_name, c)
    send_mail(subject, email, from_email, [user.email])

Given a valid email address, the password reset form does the following.

1. Lookup all users with that email address, skipping users who have unusable passwords.
2. Figure out the domain, site information, and other context data.
3. Render the password reset template.
4. Send the password reset email(s).

Our goal will be to reproduce this logic while leveraging the power of class-based views:

```python
# accounts/emails.py
class PasswordResetEmail(HTMLEmail):
    ...

send_password_reset_email = PasswordResetEmail.as_callable()
```

And to use it from a view:

```python
# accounts/views.py
from accounts.emails import send_password_reset_email

class PasswordResetView(FormView):
    ...
    def form_valid(self, form):
        # Send the password reset email.
        email = form.cleaned_data['email']
        users = UserModel._default_manager.filter(email__iexact=email)
        for user in users:
            password_reset_email(user)
        return super(PasswordResetView, self).form_valid(form)
```

Now that we know what our interface should look like, lets start writing our email class.
Step 1: Writing the basic view

First, we need a way to find all of the users who’s email matches our target email. Since we need to send a password reset email for every user with the target email, this logic needs to live outside of our email class. For this example, I’ll simply make a function to wrap around our email callable.

```python
# accounts/emails.py
from django.contrib.auth import get_user_model, UNUSABLE_PASSWORD
from emailtools import HTMLEmail

UserModel = get_user_model()

class PasswordResetEmail(HTMLEmail):
    from_address = 'admin@example.com'
    subject = 'Password reset on example.com'
    template_name = 'registration/password_reset_email.html'

    def get_to(self):
        return [self.args[0].email]

send_password_reset_email = PasswordResetEmail.as_callable()
```

Step 2: Domain and Site information.

Now let’s get our site and domain information, along with the other context information ready for template rendering. For this, we’ll want to hook into the method call to get_context_data():

```python
# accounts/emails.py
from django.contrib.auth.tokens import default_token_generator
from django.utils.http import int_to_base36
from emailtools import HTMLEmail

class PasswordResetEmail(HTMLEmail):
    token_generator = default_token_generator

    def get_context_data(self, **kwargs):
        kwargs = super(PasswordResetEmail, self).get_context_data(**kwargs)
        current_site = Site.objects.get_current()
        kwargs.update({
            'site_name': current_site.name,
            'domain': current_site.domain,
            'uid': int_to_base36(user.pk),
            'email': self.args[0].email,
            'user': self.args[0],
            'token': self.token_generator.make_token(user),
        })

        return kwargs
```

While this will suffice for reproducing the behavior of save(), constructing urls in templates via string concatenation has always seemed prone to human error. Additionally, there are so many uses for email tokens so wouldn’t it be nice to have a reusable tool for sending such emails.
Step 3: Refactoring out the Re-usable components

First, let’s write `BuildAbsoluteURIMixin`, a mixin class for your email classes which provides the url reversing that returns absolute urls.

```python
# mixins.py
from django.contrib.auth.tokens import default_token_generator
from django.contrib.sites.models import Site
from django.core.urlresolvers import reverse

class BuildAbsoluteURIMixin(object):
    protocol = 'http'

    def get_domain(self):
        return Site.objects.get_current().domain

    def get_protocol(self):
        return self.protocol

    def reverse_absolute_uri(self, view_name, args=None, kwargs=None):
        location = reverse(view_name, args=args, kwargs=kwargs)
        return self.build_absolute_uri(location)

    def build_absolute_uri(self, location):
        return '{protocol}://{domain}{location}'.format(
            protocol=self.get_protocol(),
            domain=self.get_domain(),
            location=location,
        )
```

Now, let’s write a `UserTokenEmailMixin` which will provide user based token generation for our emails.

```python
# mixins.py
from django.utils.http import int_to_base36

class UserTokenEmailMixin(BuildAbsoluteURIMixin):
    UID_KWARG = 'uidb36'
    TOKEN_KWARG = 'token'

    token_generator = default_token_generator

    def get_user(self):
        return self.args[0]

    def generate_token(self, user):
        return self.token_generator.make_token(user)

    def get_uid(self, user):
        return int_to_base36(user.pk)

    def reverse_token_url(self, view_name, args=None, kwargs={}):
        kwargs.setdefault(self.UID_KWARG, self.get_uid(self.get_user()))
        kwargs.setdefault(self.TOKEN_KWARG, self.generate_token(self.get_user()))
        return self.reverse_absolute_uri(view_name, args=args, kwargs=kwargs)
```
Step 4: Bringing it all together

Now, let's rewrite `PasswordResetEmail` to make use of these new mixins.

```python
# accounts/emails.py
from django.contrib.auth.tokens import default_token_generator
from django.utils.http import int_to_base36
from emailtools import HTMLEmail
from mixins import UserTokenEmailMixin

class PasswordResetEmail(UserTokenEmailMixin, MarkdownEmail):
    from_email = 'admin@example.com'
    template_name = 'registration/password_reset_email.html'
    subject = "Password Reset"

    def get_to(self):
        return [self.get_user().email]

    def get_context_data(self, **kwargs):
        kwargs = super(PasswordResetEmail, self).get_context_data()
        user = self.get_user()
        kwargs.update({
            'user': user,
            'reset_url': self.reverse_token_url('password_reset_confirm'),
        })
        return kwargs

send_password_reset_email = PasswordResetEmail.as_callable()
```

Step 5: Re-usability

A simple pattern for requiring email verification is to remove the password fields from the signup form and send an email verification link on account creation. This has the pleasant side effect of simplifying the signup process while verifying your user's email addresses.

Class based emails really shine here. Let's look at what it would take to use our `PasswordResetEmail` class to send a welcome email.

```python
# accounts/emails.py
send_welcome_email = PasswordResetEmail.as_callable(
    subject='Welcome to example.com'
    template_name='registration/welcome_email.html',
)
```

The two mixins found in this example are also available in email tools.
0.2.2 (2014-07-04)

• Bugfix: actually attach attachments
• Bugfix: actually use the headers parameter

0.2.1 (2013-10-30)

• make from_email default to settings.DEFAULT_FROM_EMAIL

0.2.0 (2013-09-12)

• remove message method on email callables in favor of direct class instantiation.
• make as_callable use an subclass of the email message with the overridden **kwargs calling arguments.

0.1.1 (2013-07-27)

Initial Release
Development

Development for django-emailtools happens on GitHub. Pull requests are welcome. Continuous integration is hosted on Travis CI.
Index

A
as_callable() (emailtools.cbe.base.BaseEmail method), 11

B
BaseEmail (class in emailtools.cbe.base), 11
BasicEmail (class in emailtools.cbe.base), 12

E
e-mail_message_class (emailtools.cbe.base.BaseEmail attribute), 11
e-mail_message_class (emailtools.cbe.base.BasicEmail attribute), 12
e-mail_message_class (emailtools.cbe.base.HTMLEmail attribute), 13

G
get_context_data() (emailtools.cbe.base.HTMLEmail method), 13
get_context_data() (emailtools.cbe.base.MarkdownEmail method), 13
get_email_message() (emailtools.cbe.base.BaseEmail method), 11
get_email_message_class() (emailtools.cbe.base.BaseEmail method), 11
get_email_message_kwargs() (emailtools.cbe.base.BaseEmail method), 11
get_layout_context_data() (emailtools.cbe.base.MarkdownEmail method), 13
get_layout_template() (emailtools.cbe.base.MarkdownEmail method), 13
get_send_kwargs() (emailtools.cbe.base.BaseEmail method), 11

H
HTMLEmail (class in emailtools.cbe.base), 13

L
layout_template (emailtools.cbe.base.MarkdownEmail attribute), 13

M
MarkdownEmail (class in emailtools.cbe.base), 13

S
send() (emailtools.cbe.base.BaseEmail method), 11

T
template_name (emailtools.cbe.base.HTMLEmail attribute), 13
template_name (emailtools.cbe.base.MarkdownEmail attribute), 13

25