django-mail-factory Documentation

Release 0.21

Rémy HUBSCHER

Sep 20, 2018

Contents

Django Mail Factory is a little Django app that let's you manage emails for your project very easily.

CHAPTER 1

Features

Django Mail Factory has support for:

- Multilingual
- Administration to preview or render emails
- Multi-alternatives emails: text and html
- Attachments
- HTML inline display of attached images

CHAPTER 2

Other resources

Fork it on: http://github.com/peopledoc/django-mail-factory/

Documentation: http://django-mail-factory.rtfd.org/

CHAPTER 3

Get started

From PyPI:

```
pip install django-mail-factory
```

From the github tree:

```
pip install -e http://github.com/peopledoc/django-mail-factory/
```

Then add `mail_factory` to your *INSTALLED_APPS*:

```
INSTALLED_APPS = (
    ...
    'mail_factory',
    ...
)
```

CHAPTER 4

Create your first mail

my_app/mails.py:

```
from mail_factory import factory
from mail_factory.mails import BaseMail

class WelcomeEmail(BaseMail):
    template_name = 'activation_email'
    params = ['user', 'site_name', 'site_url']

factory.register(WelcomeEmail)
```

Then you must also create the templates:

- templates/mails/activation_email/subject.txt

```
[{{site_name }}] Dear {{ user.first_name }}, your account is created
```

- templates/mails/activation_email/body.txt

```
Dear {{ user.first_name }},

Your account has been created for the site {{ site_name }}, and is
available at {{ site_url }}.

See you there soon!


The awesome {{ site_name }} team
```

- templates/mails/activation_email/body.html (optional)

CHAPTER 5

Send a mail

Using the factory:

```
from mail_factory import factory


factory.mail('activation_email', [user.email],
             {'user': user,
              'site_name': settings.SITE_NAME,
              'site_url': settings.SITE_URL})
```

Using the mail class:

```
from my_app.mails import WelcomeEmail


msg = WelcomeEmail({'user': user,
                    'site_name': settings.SITE_NAME,
                    'site_url': settings.SITE_URL})
msg.send([user.email])
```

CHAPTER 6

How does it work?

At startup, all `mails.py` files in your application folders are automatically discovered and the emails are registered to the factory.

You can then directly call your emails from the factory with their `template_name`.

It also allows you to list your emails in the administration, preview and test them by sending them to a custom address with a custom context.

Note: *mail_factory* automatically performs autodiscovery of mails modules in installed applications. To prevent it, change your INSTALLED_APPS to contain 'mail_factory.SimpleMailFactoryConfig' instead of 'mail_factory'.

This is only available in Django 1.7 and above.

CHAPTER 7

Contents

7.1 Django default mail integration

If you use Django Mail Factory, you will definitly want to manage all your application mails from Django Mail Factory.

Even if your are using some Django generic views that send mails.

7.1.1 Password Reset Mail

Here is an example of how you can use Mail Factory with the `django.contrib.auth.views.password_reset` view.

You can first add this pattern in your `urls.py`:

```
url(_(r'^password_reset/$'),
    'mail_factory.contrib.auth.views.password_reset'),
url(_(r'^password_reset/(?P<uidb36>[0-9A-Za-z]{1,13})-(?P<token>[0-9A-Za-z]{1,13}-'
      r'[0-9A-Za-z]{1,20})/$'),
    'django.contrib.auth.views.password_reset_confirm')
url(_(r'^password_reset/done/$'),
    'django.contrib.auth.views.password_reset_done')
```

Then you can overload the default templates `mails/password_reset/subject.txt` and `mails/password_reset/body.txt`.

But you can also register your own `PasswordResetMail`:

```
from django.conf import settings
from mail_factory import factory
from mail_factory.contrib.auth.mails import PasswordResetMail
from myapp.mails import AppBaseMail, AppBaseMailForm

class PasswordResetMail(AppBaseMail, PasswordResetMail):
    """Add the App header + i18n for PasswordResetMail."""
```

(continues on next page)

(continued from previous page)

```
class PasswordResetForm(AppBaseMailForm):
    class Meta:
        mail_class = PasswordResetMail
        initial = {'email': settings.ADMINS[0][1],
                   'domain': settings.SITE_URL.split('/')[2],
                   'site_name': settings.SITE_NAME,
                   'uid': u'4',
                   'user': 4,
                   'token': '3gg-37af4e5097565a629f2e',
                   'protocol': settings.SITE_URL.split('/')[0].rstrip(':')}


factory.register(PasswordResetMail, PasswordResetForm)
```

You can then update your urls.py to use this new form:

```
url(_(r'^password_reset/$'),
    'mail_factory.contrib.auth.views.password_reset',
    {'email_template_name': 'password_reset'}),
```

The default PasswordResetMail is not registered in the factory so that people that don't use it are not disturb.

If you want to use it as is, you can just register it in your app `mails.py` file like that:

```
from mail_factory import factory
from mail_factory.contrib.auth.mails import PasswordResetMail

factory.register(PasswordResetMail)
```

7.2 Hacking MailFactory

MailFactory is a tool to help you manage your emails in the real world.

That means that for the same email, regarding the context, you may want a different branding, different language, etc.

7.2.1 Specify the language for your emails

If you need to specify the language for your email, other than the currently used language, you can do so by overriding the `get_language` method on your custom class.

Let's say that our user has a `language_code` as a profile attribute:

```
class ActivationEmail(BaseMail):
    template_name = 'activation'
    params = ['user', 'activation_key']

    def get_language(self):
        return self.context['user'].get_profile().language_code
```

7.2.2 Force a param for all emails

You can also overriding the `get_params` method of a custom ancestor class to add some mandatory parameters for all your emails:

```
class MyProjectBaseMail(BaseMail):

    def get_params(self):
        params = super(MyProjectBaseMail, self).get_params()
        return params.append('user')

class ActivationEmail(MyProjectBaseMail):
    template_name = 'activation'
    params = ['activation_key']
```

This way, all your emails will have the user in the context by default.

7.2.3 Add context data

If you have some information that must be added to every email context, you can put them here:

```
class MyProjectBaseMail(BaseMail):

    def get_context_data(self, **kwargs):
        data = super(MyProjectBaseMail, self).get_context_data(**kwargs)
        data['site_name'] = settings.SITE_NAME
        data['site_url'] = settings.SITE_URL
        return data
```

7.2.4 Add attachments

Same thing here, if your branding needs a logo or a header in every emails, you can define it here:

```
from django.contrib.staticfiles import finders

class MyProjectBaseMail(BaseMail):

    def get_attachments(self, files=None):
        attach = super(MyProjectBaseMail, self).get_attachments(files)
        attach.append((finders.find('mails/header.png'),
                       'header.png', 'image/png'))
        return attach
```

Now, if you want to use this attached image in your html template, you need to use the cid URI scheme with the name of the attachment, which is the second item of the tuple (`header.png` in our example above):

```
<img src="cid:header.png" alt="This is the header" />
```

7.2.5 Template loading

By default, the template parts will be searched in:

- `templates/mails/TEMPLATE_NAME/LANGUAGE_CODE/`

- templates/mails/TEMPLATE_NAME/

But you may want to search in different locations, ie:

- templates/SITE_DOMAIN/mails/TEMPLATE_NAME/

To do that, you can override the get_template_part method:

```
class ActivationEmail(BaseMail):
    template_name = 'activation'
    params = ['activation_key', 'site']

    def get_template_part(self, part):
        """Return a mail part (body, html body or subject) template

        Try in order:

        1/ domain specific localized:
            example.com/mails/activation/fr/
        2/ domain specific:
            example.com/mails/activation/
        3/ default localized:
            mails/activation/fr/
        4/ fallback:
            mails/activation/

        """
        templates = []

        site = self.context['site']
        # 1/ {{ domain_name }}/mails/{{ template_name }}/{{ language_code}}/
        templates.append(path.join(site.domain,
                                   'mails',
                                   self.template_name,
                                   self.lang,
                                   part))
        # 2/ {{ domain_name }}/mails/{{ template_name }}/
        templates.append(path.join(site.domain,
                                   'mails',
                                   self.template_name,
                                   part))
        # 3/ and 4/ provided by the base class
        base_temps = super(MyProjectBaseMail, self).get_template_part(part)
        return templates + base_temps
```

get_template_part returns a list of template and will take the first one available.

7.3 Mail templates

When you want a multi-alternatives email, you need to provide a subject, the text/plain body and the text/html body.

All these parts are loaded from your email template directory.

templates/mails/invitation/subject.txt:

```
{% load i18n %}{% blocktrans %}[{{ site_name }}] Invitation to the beta{%␣
↪endblocktrans %}
```

A little warning: the subject needs to be on a single line

You can also create a different subject file for each language:

`templates/mails/invitation/en/subject.txt`:

```
[{{ site_name }}] Invitation to the beta
```

`templates/mails/invitation/body.txt`:

```
{% load i18n %}{% blocktrans with full_name=user.get_full_name expiration_
↪date=expiration_date|date:"l d F Y" %}
Dear {{ full_name }},

You just received an invitation to connect to our beta program.

Please click on the link below to activate your account:

{{ activation_url }}

This link will expire on: {{ expiration_date }}

{{ site_name }}
--------------------------------
If you need help for any purpose, please contact us at {{ support_email }}
{% endblocktrans %}
```

If you don't provide a `body.html` the mail will be sent in `text/plain` only, if it is present, it will be added as an alternative and displayed if the user's mail client handles html emails.

`templates/mails/invitation/body.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
↪html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>{{ site_name }}</title>
</head>
<body>
    <p><img src="cid:header.png" alt="{{ site_name }}" /></p>
    <h1>{% trans 'Invitation to the beta' %}</h1>
    <p>{% blocktrans with full_name=user.get_full_name %}Dear {{ full_name }},{%␣
↪endblocktrans %}</p>
    <p>{% trans "You just received an invitation to connect to our beta program:" %}</
↪p>
    <p>{% trans 'Please click on the link below to activate your account:' %}</p>
    <p><a href="{{ activation_url }}" target="_blank">{{ activation_url }}</a></p>
    <p>{{ site_name }}</p>
    <p>{% blocktrans %}If you need help for any purpose, please contact us at
        <a href="mailto:{{ support_email }}">{{ support_email }}</a>{% endblocktrans
↪%}</p>
</body>
</html>
```

7.4 The MailFactory Interface

In daily work email branding is important because it will call your customer to action.

MailFactory comes with a little tool that helps you to test your emails.

To do that, we generate a Django Form so that you may provide a context for your email.

Here's how you can customize your administration form.

7.4.1 Enabling MailFactory administration

You just need to enable the urls:

`project/urls.py`:

```
urlpatterns = (
    # ...
    url(r'^admin/mails/', include('mail_factory.urls')),
    url(r'^admin/', include(admin.site.urls)),
    # ...
)
```

Then you can connect to /admin/mails/ to try out your emails.

7.4.2 Registering a specific form

These two calls are equivalent:

```
from mail_factory import factory, BaseMail


class InvitationMail(BaseMail):
    template_name = "invitation"
    params = ['user']

factory.register(InvitationMail)
```

```
from mail_factory import factory, MailForm, BaseMail


class InvitationMail(BaseMail):
    template_name = "invitation"
    params = ['user']

factory.register(InvitationMail, MailForm)
```

7.4.3 Creating a custom MailForm

We may also want to build a very specific form for our email.

Let's say we have a *share this page* email, with a custom message:

```
from mail_factory import factory, BaseMail, MailForm


class SharePageMail(BaseMail):
    template_name = "share_page"
    params = ['user', 'message', 'date']


class SharePageMailForm(MailForm):
    user = forms.ModelChoiceField(queryset=User.objects.all())
    message = forms.CharField(widget=forms.Textarea)
    date = forms.DateTimeField()


factory.register(SharePageMail, SharePageMailForm)
```

7.4.4 Define form initial data

You can define `Meta.initial` to automatically provide a context for your mail.

```
import datetime
import uuid

from django.conf import settings
from django.core.urlresolvers import reverse_lazy as reverse
from django import forms
from mail_factory import factory, MailForm, BaseMail


class ShareBucketMail(BaseMail):
    template_name = 'share_bucket'
    params = ['first_name', 'last_name', 'comment', 'expiration_date',
              'activation_url']


def activation_url():
    return '%s%s' % (
        settings.SITE_URL, reverse('share:index',
                                   args=[str(uuid.uuid4()).replace('-', '')]))


class ShareBucketForm(MailForm):
    expiration_date = forms.DateField()

    class Meta:
        initial = {'first_name': 'Thibaut',
                   'last_name': 'Dupont',
                   'comment': 'I shared with you documents we talked about.',
                   'expiration_date': datetime.date.today,
                   'activation_url': activation_url}

factory.register(ShareBucketMail, ShareBucketForm)
```

Then the mail form will be autopopulated with this data.

7.4.5 Creating your application custom MailForm

By default, all email params are represented as a `forms.CharField()`, which uses a basic text input.

Let's create a project wide `BaseMailForm` that uses a `ModelChoiceField` on `auth.models.User` each time a `user` param is needed in the email.

```
from django.contrib.auth.models import User
from django import forms
from mail_factory.forms import MailForm


class BaseMailForm(MailForm):
    def get_field_for_param(self, param):
        if param == 'user':
            return forms.ModelChoiceField(
                queryset=User.objects.order_by('last_name', 'first_name'))

        return super(BaseMailForm, self).get_field_for_param(param)
```

Now you need to inherit from this `BaseMailForm` to make use of it for your custom mail forms:

```
class MyCustomMailForm(BaseMailForm):
    # your own customizations here
```

If you want this `BaseMailForm` to be used automatically when registering a mail with no custom form, here's how to do it:

```
from mail_factory import MailFactory


class BaseMailFactory(MailFactory):
    mail_form = BaseMailForm
factory = BaseMailFactory
```

And use this new factory everywhere in your code instead of `mail_factory.factory`.

7.4.6 Previewing your email

Sometimes however, you don't need or want to **render** the email, having to provide some real data (eg a user, a site, some complex model...).

The emails may be written by your sales or marketing team, set up by your designer, and all of those don't want to cope with the setting up of real data.

All they want is to be able to preview the email, in the different languages available.

This is where email **previewing** is useful.

Previewing is available straight away thanks to sane defaults. It uses the data returned by `get_preview_data` to add (possibly) non valid data to the context used to preview the mail.

This data will override any data that was returned by `get_context_data`, which in turn uses the form's Meta.initial, and in last resort, returns "###".

The preview can thus use fake data: let's take the second example from this page, the `SharePageMail`:

```
import datetime

from django.contrib.auth.models import User
from django.conf import settings

from mail_factory import factory, MailForm


class SharePageMailForm(MailForm):
    user = forms.ModelChoiceField(queryset=User.objects.all())
    message = forms.CharField(widget=forms.Textarea)
    date = forms.DateTimeField()

    class Meta:
        initial = {'message': 'Some message'}

    def get_preview_data(self, **kwargs):
        data = super(SharePageMailForm, self).get_preview_data(**kwargs)
        data['date'] = datetime.date.today()
        # create on-the-fly fake User, not saved in database: not valid data
        # but still added to context for previewing
        data['user'] = User(first_name='John', last_name='Doe')
        return data

factory.register(SharePageMail, SharePageMailForm)
```

With this feature, when displaying the mail form in the admin (to render the email with real data), the email will also be previewed in the different available languages with the fake data provided by the form's `get_preview_data`, which overrides the data returned by `get_context_data`.