django-classymail Documentation

Release 1.0-alpha

Rafal Stozek

March 18, 2013

CONTENTS

It's easy to send a simple e-mail using Django. But e-mails sometimes gets really complicated and un-DRY. That's why I created ClassyMail.

CHAPTER

ONE

FEATURES

- Timezone and language support
- CSS inlining support - allows you to write normal stylesheets instead of `style=""` attributes for every tag
- Code reuse with mixins (just like class based views)

CHAPTER

TWO

INSTALLATION

ClassyMail is available on PyPI, so just install using pip:

```
pip install django-classymail
```

Contents:

2.1 Quickstart

Django makes it really easy to send e-mail messages:

```
from django.core.mail import EmailMessage

msg = EmailMessage(subject="Hello", body="Hello there!", to=['me@example.com'])
msg.send()
```

But this code will soon get really complicated. For example - we should send both plain text version (for old or simple e-mail clients) and html version of an e-mail:

```
from django.core.mail import EmailMultiAlternatives

subject, from_email, to = 'hello', 'from@example.com', '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()
```

This code will soon get even more complicated, because you need to also worry about things like:

- More meaningful (dynamic) topic
- Which language and timezone should be used when rendering templates? We also should use them when generating e-mail's subject.
- Inlining css to make writing html e-mail templates easier
- Generating full urls for e-mails (not just paths)
- Attachments
- Code reuse

2.1.1 Basics

ClassyMail uses concept similar to django class based views. You should be already familiar with it.

Just like with CBVs (Class Based Views) you can set attributes on class that will be used to build an e-mail:

```
from django.utils.translation import ugettext_lazy as _
from classymail import ClassyMail

class WelcomeMail(ClassyMail):
    subject = _("Welcome to our website. Thanks for joining us!")

builder = WelcomeMail()
# prints 'Welcome to our website. Thanks for joining us!'
print(builder.subject)
```

Also - just like with CBVs you can easily override those attributes using keyword arguments to class. The only difference is that there is no `.as_view()` method:

```
builder = WelcomeMail(subject="We can override subject", to=['me@example.com'])
print(builder.subject) # prints 'We can override subject'
print(builder.to) # prints ['me@example.com']
```

Both `subject` and `to` are attributes of `ClassyMail` parent class. `subject` defaults to empty string and `to` defaults to `None`. You can override them using keyword arguments to your e-mail class like on the example above.

But sometimes just setting attribute on a class is not enough. Sometimes we need things to be more dynamic. When you subclass ClassyMail you can override attributes like `subject` or methods like `get_subject`:

```
class WelcomeMail(ClassyMail):
    user = None

    def get_subject(self):
        return _("Welcome %s! Thanks for joining us!") % self.user.first_name
```

2.1.2 Html and text content

Now that we know how to set a proper topic for our e-mail message let's set html and text template names to create our first real, working e-mail:

```
class WelcomeMail(ClassyMail):
    user = None
    html_template_name = 'emails/welcome.html'
    text_template_name = 'emails/welcome.txt'

    def get_subject(self):
        return _("Welcome %s! Thanks for joining us!") % self.user.first_name

    # there's also get_context_data() similar to one in CBVs
    def get_context_data(self):
        data = super(WelcomeMail, self).get_context_data()
        data['user'] = self.user
        return data
```

Now we have to prepare templates. If you have ever sent an html e-mail using django you know that you can put your styles only in `style=""` attributes because `<style>` tags are ignored.

That's not true for ClassyMail. Behind the scenes ClassyMail will find all styles in `<style>` tags and put them inside `style=""` attribute for you. For example an e-mail template like this:

```
<html>
<head>
    <style>
    h1 { background: #eee; color: #333; }
    div.footer .who { font-style: italic; }
    div.footer .awesome { font-weight: bold; }
    </style>
</head>
<body>
    <h1>Welcome {{ user.first_name }}!</h1>
    <p>We are very happy that you decided to join us.</p>

    <div class="footer">
        Best,<br>
        <span class="who">
            Our <span class="awesome">awesome</span> team.
        </span>
    </div>
</body>
</html>
```

will result in e-mail with contents like this:

```
<html>
<head></head>
<body style="padding:10px">
    <h1 style="color:#333; background:#eee">Welcome Matt!</h1>
    <p>We are very happy that you decided to join us.</p>

    <div>
        Best,<br><span style="font-style:italic">
            Our <span style="font-weight:bold">awesome</span> team.
        </span>
    </div>
</body>
</html>
```

that's helpful, isn't it?

2.1.3 Sending e-mails

Now that we have a real welcome e-mail class we should send it:

```
builder = WelcomeMail(to=[user.email])
# build() builds instance of django.core.mail.EmailMessage
msg = builder.build()
msg.send()
```

As every keyword argument - `to` is just another attribute on ClassyMail class that is being overridden. `build()` method returns new `django.core.mail.EmailMessage` instance, which we can send.

Boy, 3 lines of code. That's a lot of boilerplate. Actually there is a shortcut, but example above shows how actually ClassyMail works - ClassyMail subclasses are just builders which build instances of `EmailMessage` classes.

Now that you know how this actually works you can use convenient shortcut:

```
WelcomeMail.send(to=[user.email])
```

But remember - if you're sending more than just one e-mail message then you may want to build your messages and then send them using single connection. See how to send multiple e-mails.

2.1.4 Timezone and language

When sending an e-mails to user we should use timezone and language preferred by user rather than one active for current request. ClassyMail helps you with that by providing `timezone` and `language` attributes and `get_timezone()` and `get_language()` methods which you can override.

```
class WelcomeMail(ClassyMail):
    user = None
    html_template_name = 'emails/welcome.html'
    text_template_name = 'emails/welcome.txt'
    timezone = pytz.timezone("Europe/Warsaw")

    def get_subject(self):
        return _("Welcome %s! Thanks for joining us!") % self.user.first_name

    def get_language(self):
        return self.user.get_profile().language
```

In the example above we infer language from user's profile. Both language and timezone will be used not just to render templates but also e-mail's subject (actually all `get_*` methods are called with changed language and timezone).

2.1.5 Reusing code

Don't repeat yourself. ClassyMail achieves this - like CBVs - using mixins. Let's see an example of a simple mixin:

```
class UserMixin(ClassyMail):
    """
    Sets language and timezone according to user preferences, adds "user" to
    template context and sets recipient to user's email address.
    """
    user = None

    def get_timezone(self):
        return self.user.get_profile().timezone

    def get_language(self):
        return self.user.get_profile().language

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

    def get_context_data(self):
        data = super(UserMixin, self).get_context_data()
        data['user'] = self.user
        return data
```

This little piece of code will infer timezone and language from "user" attribute, add "user" to template context and set recipient to user's email. It is also easy to test and will make our welcome email class a lot simpler.

```
class WelcomeEmail(UserMixin, ClassyMail):
    html_template_name = 'emails/welcome.html'
    text_template_name = 'emails/welcome.txt'
```

```
    def get_subject(self):
        return _("Welcome %s! Thanks for joining us!") % self.user.first_name
```

Not only our e-mail logic is now simpler because we can just include UserMixin but also we can write tests for user timezone, language etc. once.

Note: When subclassing you should always place mixins before base classes. This will help you avoid problems with MRO (method resolution order).
