django-simple-sms Documentation

Release 1.0.0

Thibault Jouannic

December 05, 2014

Contents

django-simple-sms is a Django application to easily send text messages from your Django application.

CHAPTER 1

Philosophy

The application intends to give you an easy and quick yet reliable way to send text messages using a third party text message carrier.

CHAPTER 2

Compatibility

The application is compatible and tested against the two latest Django versions (1.6 and 1.7) and the two main Python versions (2.7 and 3.4).

CHAPTER 3

Example usage

```
# -*- coding: utf-8 -*-

from __future__ import unicode_literals

from djsms import send_text


frm = '+33123456789'
to = '+33987654321'
text = 'Please remember to pick up the bread before coming'
send_text(text, frm, to)
```

CHAPTER 4

Contents

4.1 Installation

Install the application with pip:

```
pip install django-simple-sms
```

Add *djsms* to you `INSTALLED_APPS`:

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

Set a *text message backend* in your settings:

```
DJSMS_BACKEND = 'djsms.backends.FileBasedBackend'
```

4.2 Sending text messages

To send text messages, you can use a high-level helper api or use the low level api.

4.2.1 Quick example

```
# -*- coding: utf-8 -*-

from __future__ import unicode_literals

from djsms import send_text


frm = '+33123456789'
to = '+33987654321'
text = 'Please remember to pick up the bread before coming'
message = send_text(text, frm, to)
```

4.2.2 The `send_text` method

Note: Phone numbers must be formatted in international format, with the country code at the beginning and a leading "+".

The easiest way to send text messages is to use the `send_text` utility method. It's a convenient helper to quickly access the application features.

`djsms.`**`send_text`**(*text*, *frm*, *to*, *fail_silently=True*, *status_report=False*)

A convenient helper to quickly access the application features.

Returns a `TextMessage` object.

Required parameters:

Parameters

- **text** – the text message body. It's length is not limited, but if it's too long, your carrier will probably split it and send in a multipart text message, and you will be charged accordingly
- **frm** – the originator phone number. Some carriers accept alphanumerical values, but it really depends on mobile networks and local laws.
- **to** – the recipient phone number.

Additionals parameters:

Parameters

- **fail_silently** – If True, errors will just be ignored.
- **status_report** – If True, asks the selected carrier to provide status report by asynchronously calling an uri. If your carrier does'nt provide this option, the parameter value will simply be ignored.

Raises:

Any `TextMessageError` subclass if `fail_silently` is False.

Warning: Most backends will work using a REST Api. `send_text` will result in a blocking HTTP request which can generate a noticeable delay if called during a client's request processing.
You might want to delegate the actual call to the method in a Celery task.

4.2.3 The `TextMessage` class

class `djsms.models.`**`TextMessage`**(**args*, ***kwargs*)

An outgoing or incoming text message representation.

Note: Messages sent as multipart are still represented by a single `TextMessage` object.

`TextMessage` objects will be created by other part of the app (e.g the `send_text` method or incoming messages), so you don't have to create them by yourself.

You can however access `TextMessage` objects to have information about sent and received messages, such as price, status, date of creation, etc.

The following attributes are available:

•*frm*: the sender's phone number

- *to*: the recipient's phone number
- *text* : the message's content
- *direction*: flag to show whether it's an incoming or outgoing message. Values are 'incoming' or 'outgoing'.
- *price (Decimal)*: the price billed by your provider for this message
- *status*: the message's current status
- *created_on*: the date / time of the message creation

Example usage:

```
>>> message = send_text(text, frm, to)
>>> print message.price
0.015
>>> print message.direction
'outgoing'
>>> print message.status
'sent'
```

Warning: The `price` attributes contains the price charged by the provider for this text message. This price can be expressed in any currency, however, depending on your account configuration. Check your provider documentation to know more.

STATUSES

The list of different possible statuses.

- *created*: the message was just created in the database and was not yet submitted to your text message provider
- *sending*: the message was successfully sent to your provider
- *sent*: the message was successfully sent to the upstream carrier
- *delivered*: the message was successfully delivered to the destination handset
- *refused*: the message was refused by your provider
- *rejected*: the message was refused by the destination carrier

Warning: Because a message was received and accepted by your text message provider does not mean it will be accepted by the destination carrier, nor the destination handset. Be careful not to be confused by the different statuses.

4.3 Getting status reports

Some text message providers can asynchronously call one of your api to provide details about message delivery statuses.

4.3.1 Declaring the callback uri

The application already defines the correct callback uris. All you have to do is import them in your main's `urls.py` file.

```
urlpatterns = patterns('',
    ...
    url(r'^djsms', include('djsms.urls')),
    ...
)
```

4.3.2 Require status reports

On most text message carriers, the status report is a per-message option, so we kept it that way in the application. To require status updates for a given message, you need to pass the correct option to `send_text()`:

```
message = send_text(text, frm, to, status_report=True)
```

Some carriers require that you set the callback uri directly in their dashboard. Check for your provider documentation. The callback uri will be:

```
http://your-site.com/djsms/callback/
```

4.3.3 Reacting to status updates

Internally, the callback view updates the corresponding `TextMessage` objects. If you need to perform actions on those updates, you can declare signals in one of your own applicatin. Here is a quick example.

```
# -*- coding: utf-8 -*-

from __future__ import unicode_literals

from django.db.models.signals import post_save
from django.dispatch import receiver
from djsms.models import TextMessage

from notifications.models import Notification


@receiver(post_save, sender=TextMessage, dispatch_uid='message_update_user')
def update_notification(sender, **kwargs):
    """Update ``Notification`` object when the text message was sent."""
    message = kwargs.get('instance')
    created = kwargs.get('created')

    if not created:
        Notification.object \
            .filter(message=message) \
            .update(status=message.status)
```

4.4 Receiving messages

Some providers will allow for incoming text messages. Upon receiving a message, they will call an api of yours.

4.4.1 Declaring the incoming uri

For accepting incoming text messages, you must import the application views.

```
urlpatterns = patterns('',
    ...
    url(r'^djsms', include('djsms.urls')),
    ...
)
```

A new uri will be available to accept incoming messages:

```
http://your-site.com/djsms/incoming/
```

Check your text message provider configuration to know how to configure this uri.

4.4.2 Reacting to incoming messages

When a new text messages is received, a new `TextMessage` will be created. To react to this, you can use Django signals.

Here is a quick example.

```
# -*- coding: utf-8 -*-

from __future__ import unicode_literals

from django.db.models.signals import post_save
from django.dispatch import receiver
from djsms.models import TextMessage

from notifications.models import Notification


@receiver(post_save, sender=TextMessage, dispatch_uid='message_create_notification')
def create_notification(sender, **kwargs):
    """Create a new ``Notification`` object when a text message is received."""
    message = kwargs.get('instance')
    created = kwargs.get('created')

    if created:
        Notification.objects.create(text_message=message)
```

4.5 Text message backends

The application can be configured to use different backends to send the actual text messages. Some backends are only used for testing and not intended for production.

4.5.1 Dummy backend

class djsms.backends.**DummyBackend**

As it names says, it's a dummy backend.

This backend does nothing. It exists because it was easy to write :) Just kidding... It can be helpful during development phase, but it's obviously useless in production.

```
DJSMS_BACKEND = 'djsms.backends.DummyBackend'
```

4.5.2 Console backend

class djsms.backends.**ConsoleBackend**
Sends text messages to the console.

Outputs sent messages to the console. It can be convenient during development, but is not intended for production.

```
DJSMS_BACKEND = 'djsms.backends.ConsoleBackend'
```

4.5.3 Nexmo Backend

class djsms.backends.**NexmoBackend**
Uses the Nexmo provider.

The Nexmo Backend uses the Nexmo provider for outgoing messages. Internally, the application uses libnexmo to communicate with the Nexmo API

Did I mention that you need an existing Nexmo account? Isn't it obvious?

```
NEXMO_API_KEY = 'api_key'
NEXMO_API_SECRET = 'api_secret'
DJSMS_BACKEND = 'djsms.backends.NexmoBackend'
```

4.5.4 Define your own backend

It's easy to write your own backend. All you have to do is override the `BaseBackend` class.

class djsms.backends.**BaseBackend**
The base class for every backends.

4.6 Errors and exceptions

Sending text messages is a tricky job. Your python code will issue an http request to a text message provider, that will dispatch your message to a carrier that will dispatch it to the end user handheld. A lot of things could go wrong.

Hence we define different error classes to handle errors as accurately as possible.

However, remember that you can never be 100% sure that a given message was actually read by it's intented recipient.

4.6.1 Exceptions

exception djsms.exceptions.**TextMessageError**
Base exception for all text message errors.

4.7 Colophon

This application was written with love by Thibault Jouannic. I'm available for Python / Django and / or Javascript freelance missions.

You can also follow me on @twitter.

d

B

C

D

N

S

T