django-sitemessage Documentation

Release 1.4.0

Igor ‘idle sign‘ Starikov

Mar 18, 2023

Contents

https://github.com/idlesign/django-sitemessage

CHAPTER 1

Description

Reusable application for Django introducing a message delivery framework.

Features:

- **Message Types** - message classes exposing message composition logic (plain text, html, etc.).
- **Messengers** - clients for various protocols (smtp, jabber, twitter, telegram, facebook, vkontakte, etc.);
- Support for user defined message types.
- Support for user defined messenger types.
- Message prioritization.
- Message subscription/unsubscription system.
- Message grouping to prevent flooding.
- Message 'read' indication.
- Means for background message delivery and cleanup.
- Means to debug integration: test requisites, delivery log.
- Django Admin integration.

Currently supported messengers:

1. SMTP;
2. XMPP (requires `sleekxmpp` package);
3. Twitter (requires `twitter` package);
4. Telegram (requires `requests` package);
5. Facebook (requires `requests` package);
6. VKontakte (requires `requests` package).

CHAPTER 2

Requirements

1. Python 3.7+
2. Django 2.0+

CHAPTER 3

Table of Contents

3.1 Quickstart

- Add the **sitemessage** application to INSTALLED_APPS in your settings file (usually 'settings.py').
- Run './manage.py syncdb' to install *sitemessage* tables into database.

Note: When switching from an older version do not forget to upgrade your database schema.

That could be done with the following command issued in your Django project directory:

```
./manage.py migrate
```

1. Configure messengers for your project (create `sitemessages.py` in one of your apps):

```
from sitemessage.toolbox import register_messenger_objects
from sitemessage.messengers.smtp import SMTPMessenger
from sitemessage.messengers.xmpp import XMPPSleekMessenger


# We register two messengers to deliver emails and jabber messages.
register_messenger_objects(
    SMTPMessenger('user1@host.com', 'user1', 'user1password', host='smtp.
↪host.com', use_tls=True),
    XMPPSleekMessenger('user1@jabber.host.com', 'user1password', 'jabber.
↪host.com'),
)

# Or you may want to define your own message type for further usage.
class MyMessage(MessageBase):

    alias = 'myxmpp'
    supported_messengers = ['xmppsleek']
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def create(cls, message: str):
        cls(message).schedule(cls.recipients('xmppsleek', ['a@some.tld',
↪'b@some.tld', ]))

register_message_types(MyMessage)
```

2. Schedule messages for delivery when and where needed (e.g. in a view):

```
from sitemessage.shortcuts import schedule_email, schedule_jabber_message
from .sitemessages import MyFbMessage


def send_messages_view(request):
    ...
    # Suppose `user_model` is a recipient User Model instance.
    user1_model = ...

    # Schedule both email and jabber messages to delivery.
    schedule_email('Email from sitemessage.', [user1_model, 'user2@host.
↪com'])
    schedule_jabber_message('Jabber message from sitetree', [
↪'user1@jabber.host.com', 'user2@jabber.host.com'])
    ...

    # Or if you want to send your message type:
    MyMessage.create('Hi there!')
```

3. Periodically run Django management command from wherever you like (cli, cron, Celery, uWSGI, etc.):

```
./manage.py sitemessage_send_scheduled
```

3.2 Toolbox

sitemessage toolbox exposes some commonly used functions.

3.2.1 Defining message recipients

sitemessage.toolbox.recipients allows to define message recipients for various messengers, so that they could be passed into message scheduling functions:

```
from sitemessage.toolbox import recipients
from sitemessage.messengers.smtp import SMTPMessenger
from sitemessage.messengers.xmpp import XMPPSleekMessenger


# The first argument could be Messenger alias:
my_smtp_recipients = recipients('smtp', ['user1@host.com', 'user2@host.com']),

# or a Messenger class itself:
my_jabber_recipients = recipients(XMPPSleekMessenger, ['user1@jabber.host.com',
↪'user2@jabber.host.com']),
```

(continues on next page)

(continued from previous page)

```
# Second arguments accepts either Django User model instance or an actual address:
user1_model = ...
my_smtp_recipients = recipients(SMTPMessenger, [user1_model, 'user2@host.com'])

# You can also merge recipients from several messengers:
my_recipients = my_smtp_recipients + my_jabber_recipients
```

3.2.2 Scheduling messages

sitemessage.toolbox.schedule_messages is a generic tool to schedule messages:

```
from sitemessage.toolbox import schedule_messages, recipients
# Let's import a built-in message type class we'll use.
from sitemessage.messages import EmailHtmlMessage


schedule_messages(
    # You can pass one or several message objects:
    [
        # The first param of this Message Type is `subject`. The second may be either␣
↪an html itself:
        EmailHtmlMessage('Message subject 1', '<html><head></head><body>Some <b>text</
↪b></body></html>'),

        # or a dictionary
        EmailHtmlMessage('Message subject 2', {'title': 'My message', 'entry': 'Some␣
↪text.'}),

        # NOTE: Different Message Types may expect different arguments.
    ],

    # The same applies to recipients: add one or many as required:
    recipients('smtp', ['user1@host.com', 'user2@host.com']),

    # It's useful sometimes to know message sender in terms of Django users:
    sender=request.user
)
```

3.2.3 Sending test messages

When your messengers are configured you can try and send a test message using **sitemessage_probe** management command:

```
./manage.py sitemessage_probe smtp --to someone@example.com
```

Or you can use **sitemessage.toolbox.send_test_message** function:

```
from sitemessage.toolbox import send_test_message

send_test_message('smtp', to='someone@example.com')
```

3.2.4 Sending messages

Scheduled messages are normally sent with the help of **sitemessage_send_scheduled** management command, that could be issued from wherever you like (cron, Celery, etc.):

```
./manage.py sitemessage_send_scheduled
```

Nevertheless you can directly use **sitemessage.toolbox.send_scheduled_messages** from sitemessage toolbox:

```
from sitemessage.toolbox import send_scheduled_messages


# Note that this might eventually raise UnknownMessengerError,␣
↪UnknownMessageTypeError exceptions.
send_scheduled_messages()

# Or if you do not want sitemessage exceptions to be raised (that way scheduled␣
↪messages
# with unknown message types or for which messengers are not configured won't be␣
↪sent):
send_scheduled_messages(ignore_unknown_messengers=True, ignore_unknown_message_
↪types=True)

# To send only messages of a certain priority use `priority` argument.
send_scheduled_messages(priority=10)
```

3.2.5 Cleanup sent messages and dispatches

You can delete sent dispatches and message from DB using **sitemessage_cleanup**:

```
./manage.py sitemessage_cleanup --ago 5
```

Or you can use **sitemessage.toolbox.cleanup_sent_messages** from sitemessage toolbox:

```
from sitemessage.toolbox import cleanup_sent_messages

# Remove all dispatches (but not messages) 5 days old.
cleanup_sent_messages(ago=5, dispatches_only=True)

# Delete all sent messages and dispatches.
cleanup_sent_messages()
```

3.2.6 Use sitemessage to send Django-generated e-mails

In *settings.py* of your project set *EMAIL_BACKEND* to a backend shipped with **sitemessage**.

```
EMAIL_BACKEND = 'sitemessage.backends.EmailBackend'
```

After that Django's *send_mail()* function will schedule e-mails using **sitemessage** machinery.

3.3 Messages

sitemessage message classes expose message composition logic (plain text, html, etc.).

You can either use builtin classes or define your own.

3.3.1 Helper functions

- **sitemessage.toolbox.register_message_types(*message_types)**

  Registers message types (classes).

- **get_registered_message_types()**

  Returns registered message types dict indexed by their aliases.

- **get_registered_message_type(message_type)**

  Returns registered message type (class) by alias,

3.3.2 Builtin message types

Builtin message types are available from **sitemessage.messages**:

- **sitemessage.messages.plain.PlainTextMessage**
- **sitemessage.messages.email.EmailTextMessage**
- **sitemessage.messages.email.EmailHtmlMessage**

3.3.3 User defined message types

To define a message type one needs to inherit from **sitemessage.messages.base.MessageBase** (or a builtin message class), and to register it with **sitemessage.toolbox.register_message_types** (put these instructions into *sitemessages.py* in one of your apps):

```
from sitemessage.messages.base import MessageBase
from sitemessage.toolbox import register_message_types
from django.utils import timezone


class MyMessage(MessageBase):

    # Message types could be addressed by aliases.
    alias = 'mymessage'

    # Message type title to show up in UI
    title = 'Super message'

    # Define a template path to build messages from.
    # You can omit this setting and place your template under
    # `templates/sitemessage/messages/` naming it as `mymessage__<messenger>.html`
    # where <messenger> is a messenger alias, e.g. `smtp`.
    template = 'mymessages/mymessage.html'

    # Define a send retry limit for that message type.
    send_retry_limit = 10

    # If we don't want users to subscribe for messages of this type
    # (see get_user_preferences_for_ui()) we just forbid such subscriptions.
    allow_user_subscription = False
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, text, date):
        # Calling base class __init__ and passing message context
        super(MyMessage, self).__init__({'text': text, 'date': date})

    @classmethod
    def get_template_context(cls, context):
        """Here we can add some data into template context
        right before rendering.

        """
        context.update({'greeting': 'Hi!'})
        return context

    @classmethod
    def create(cls, text):
        """Let it be an alternative constructor - kind of a shortcut."""

        # This recipient list is comprised of users subscribed to this message type.
        recipients = cls.get_subscribers()

        # Or we can build recipient list for a certain messenger manually.
        # recipients = cls.recipients('smtp', 'someone@sowhere.local')

        date_now = timezone.now().date().strftime('%d.%m.%Y')
        cls(text, date_now).schedule(recipients)

register_message_types(MyMessage)
```

Note: Look through `MessageBase` and other builtin message classes for more code examples.

Now, as long as our message type uses a template, let's create it (*mymessages/mymessage.html*):

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>{{ greeting }}</title>
</head>
<body>
    <h1>{{ greeting }}</h1>
    {{ text }}
    <hr>
    {{ date }}
</body>
</html>
```

Note: The following context variables are available in templates by default:

SITE_URL - base site URL

message_model - message model data

dispatch_model - message dispatch model data

directive_unsubscribe - unsubscribe directive string (e.g. URL, command)

directive_mark_read - mark dispatch as read directive string (e.g. Url, command)

After that you can schedule and send messages of this new type:

```
from sitemessage.toolbox import schedule_messages, recipients
from myproject.sitemessages import MyMessage


# Scheduling message send via smtp.
schedule_messages(MyMessage('Some text', '17.06.2014'), recipients('smtp',
↪'user1@host.com'))

# Or we can use out shortcut method:
MyMessage.create('Some other text')
```

3.4 Messengers

sitemessage messenger classes implement clients for various protocols (smtp, jabber, etc.).

You can either use builtin classes or define your own.

3.4.1 Helper functions

- `sitemessage.toolbox.register_messenger_objects(*messengers)`

  Registers (configures) messengers.

- `sitemessage.toolbox.get_registered_messenger_objects()`

  Returns registered (configured) messengers dict indexed by messenger aliases.

- `sitemessage.toolbox.get_registered_messenger_object(messenger)`

  Returns registered (configured) messenger by alias,

3.4.2 Builtin messengers

Builtin messengers are available from **sitemessage.messengers**:

smtp.SMTPMessenger

aliased *smtp*

Warning: Uses Python's built-in `smtplib`.

xmpp.XMPPSleekMessenger

aliased *xmppsleek*

Warning: Requires `sleekxmpp` package.

```
from sitemessage.toolbox import schedule_messages, recipients


# Sending jabber message.
schedule_messages('Hello there!', recipients('xmppsleek', 'somebody@example.ru'))
```

twitter.TwitterMessenger

aliased *twitter*

Warning: Requires `twitter` package.

```
from sitemessage.toolbox import schedule_messages, recipients


# Twitting example.
schedule_messages('My tweet.', recipients('twitter', ''))

# Tweet to somebody.
schedule_messages('Hey, that is my tweet for you.', recipients('twitter', 'idlesign'))
```

telegram.TelegramMessenger

aliased *telegram*

Warning: Requires `requests` package and a registered Telegram Bot. See https://core.telegram.org/bots/api

To send messages to a channel, your bot needs to be and administrator of that channel.

```
from sitemessage.toolbox import schedule_messages, recipients


# Let's send a message to chat with ID 12345678.
# (To get chat IDs from `/start` command messages sent to our bot
# by users you can use get_chat_ids() method of Telegram messenger).
schedule_messages('Hi there!', recipients('telegram', '12345678'))

# Message to a channel mychannel
schedule_messages('Hi all!', recipients('telegram', '@mychannel'))
```

facebook.FacebookMessenger

aliased *fb*

Warning: Requires `requests` package, registered FB application and page.

See `FacebookMessenger` docstring for detailed instructions.

```
from sitemessage.toolbox import schedule_messages, recipients


# Schedule a message or a URL for FB timeline.
schedule_messages('Hi there!', recipients('fb', ''))
```

vkontakte.VKontakteMessenger

aliased *vk*

Warning: Requires `requests` package, registered VK application and community page.

See `VKontakteMessenger` docstring for detailed instructions.

```
from sitemessage.toolbox import schedule_messages, recipients


# Schedule a message or a URL for VK page wall. 1245 - user_id; use -12345 (with␣
↪minus) to post to community wall.
schedule_messages('Hi there!', recipients('vk', '12345'))
```

3.4.3 Proxying

Some messengers (*vk*, *fb*, *telegram*) are able to use proxies (e.g. SOCKS5).

One may pass *proxy* argument to use proxies.

```
TelegramMessenger('token', proxy={'https': 'socks5://user:pass@host:port'})
```

3.4.4 Sending test messages

After a messenger is configured you can try whether it works properly using its **send_test_message** method:

```
from sitemessage.messengers.smtp import SMTPMessenger


msgr = SMTPMessenger('user1@host.com', 'user1', 'user1password', host='smtp.host.com',
↪ use_tls=True)
msgr.send_test_message('user1@host.com', 'This is a test message')
```

3.4.5 User defined messengers

To define a message type one needs to inherit from **sitemessage.messengers.base.MessengerBase** (or a builtin messenger class), and to register it with **sitemessage.toolbox.register_messenger_objects** (put these instructions into *sitemessages.py* in one of your apps):

```
from sitemessage.messengers.base import MessengerBase
from sitemessage.toolbox import register_messenger_objects


class MyMessenger(MessengerBase):

    # Messengers could be addressed by aliases.
    alias = 'mymessenger'

    # Messenger title to show up in UI
    title = 'Super messenger'

    # If we don't want users to subscribe for messages from that messenger
    # (see get_user_preferences_for_ui()) we just forbid such subscriptions.
    allow_user_subscription = False

    def __init__(self):
        """This messenger doesn't accept any configuration arguments.
        Other may expect login, password, host, etc. to connect this messenger to a␣
↪service.

        """
    @classmethod
    def get_address(cls, recipient):
        address = recipient
        if hasattr(recipient, 'username'):
            # We'll simply get address from User object `username`.
            address = '%s--address' % recipient.username
        return address

    def before_send(self):
        """We don't need that for now, but usually here will be messenger warm up␣
↪(connect) code."""

    def after_send(self):
        """We don't need that for now, but usually here will be messenger cool down␣
↪(disconnect) code."""

    def send(self, message_cls, message_model, dispatch_models):
        """This is the main sending method that every messenger must implement."""

        # `dispatch_models` from sitemessage are models representing a dispatch
        # of a certain message_model for a definite addressee.
        for dispatch_model in dispatch_models:

            # For demonstration purposes we won't send a dispatch anywhere,
            # we'll just mark it as sent:
            self.mark_sent(dispatch_model)  # See also: self.mark_failed() and self.
↪mark_error().

register_messenger_objects(MyMessenger())
```

Note: Look through `MessengerBase` and other builtin messenger classes for more information and code examples.

After that you can schedule and send messages with your messenger as usual:

```
from sitemessage.toolbox import schedule_messages, recipients


user2 = ...  # Let's suppose it's an instance of Django user model.
# We'll just try to send PlainText message.
schedule_messages('Some plain text message', recipients('mymessenger', ['user1--
↪address', user2]))
```

3.5 Exceptions

The following exception classes are used by *sitemessage*.

exception sitemessage.exceptions.**MessengerException**
Base messenger exception.

exception sitemessage.exceptions.**MessengerWarmupException**
This exception represents a delivery error due to a messenger warm up process failure.

exception sitemessage.exceptions.**SiteMessageConfigurationError**
This error is raised on configuration errors.

exception sitemessage.exceptions.**SiteMessageError**
Base class for sitemessage errors.

exception sitemessage.exceptions.**UnknownMessageTypeError**
This error is raised when there's a try to access an unknown message type.

exception sitemessage.exceptions.**UnknownMessengerError**
This error is raised when there's a try to access an unknown messenger.

3.6 Prioritizing messages

sitemessage supports message sending prioritization: any message might be given a positive number to describe its priority.

Note: It's up to you to decide the meaning of priority numbers.

Prioritization is supported on the following two levels:

1. You can define *priority* within your message type class.

```
from sitemessage.messages.base import MessageBase


class MyMessage(MessageBase):

    alias = 'mymessage'

    priority = 10  # Messages of this type will automatically have priority of 10.

    ...
```

2. Or you can override priority defined within message type, by supplying *priority* argument to messages scheduling functions.

```
from sitemessage.shortcuts import schedule_email
from sitemessage.toolbox import schedule_messages, recipients


schedule_email('Email from sitemessage.', 'user2@host.com', priority=1)

# or

schedule_messages('My message', recipients('smtp', 'user1@host.com'), priority=16)
```

After that you can use **sitemessage_send_scheduled** management command with **–priority** argument to send message when needed:

```
./manage.py sitemessage_send_scheduled --priority 10
```

Note: Use a scheduler (e.g cron, uWSGI cron/cron2, etc.) to send messages with different priorities on different days or intervals, and even simultaneously.

3.7 Grouping messages

sitemessage allows you to group messages in such a way that even if your application generates many messages (between send attempts) your user receives them as one.

```
from sitemessage.messages.base import MessageBase


class MyMessage(MessageBase):

    ...

    # Define group ID at class level or as a @property
    group_mark = 'groupme'

    # In case your message has some complex context
    # you may want to override 'merge_context' to add a new message
    # context to the context already existing in message stored in DB
    @classmethod
    def merge_context(cls, context: dict, new_context: dict) -> dict:
        merged = ...  #
        return merged
```

3.8 Recipients and subscriptions

3.8.1 Postponed dispatches

Note that when scheduling a message you can omit *recipients* parameter.

In that case no dispatch objects are created on scheduling, instead the process of creation is postponed until *prepare_dispatches()* function is called.

```python
from sitemessage.toolbox import schedule_messages, prepare_dispatches

# Here `recipients` parameter is omitted ...
schedule_messages(MyMessage('Some text'))

# ... instead dispatches are created later.
prepare_dispatches()
```

prepare_dispatches() by default generates dispatches using recipients list comprised from users subscription preferences data (see below).

3.8.2 Handling subscriptions

sitemessage supports basic subscriptions mechanics, and that's how it works.

sitemessage.toolbox.get_user_preferences_for_ui is able to generate user subscription preferences data, that could be rendered in HTML as table using **sitemessage_prefs_table** template tag.

Note: **sitemessage_prefs_table** tag support table layout customization through custom templates.

- **user_prefs_table-bootstrap.html** - Bootstrap-style table.

  {% sitemessage_prefs_table from subscr_prefs template "sitemessage/user_prefs_table-bootstrap.html" %}

This table in its turn could be placed in *form* tag to allow users to choose message types they want to receive using various messengers.

At last **sitemessage.toolbox.set_user_preferences_from_request** can process *form* data from a request and store subscription data into DB.

```python
from django.shortcuts import render
from sitemessage.toolbox import set_user_preferences_from_request, get_user_
↪preferences_for_ui


def user_preferences(self, request):
    """Let's suppose this simplified view handles user preferences."""

    ...

    if request.POST:
        # Process form data:
        set_user_preferences_from_request(request)
        ...

    # Prepare preferences data.
    subscr_prefs = get_user_preferences_for_ui(request.user)

    ...

    return render(request, 'user_preferences.html', {'subscr_prefs': subscr_prefs})
```

Note: **get_user_preferences_for_ui** allows messenger titles customization and both message types and messengers filtering. You can also forbid subscriptions on message type or messenger level by setting *allow_user_subscription* class attribute to *False*.

And that's what is in a template used by the view above:

```
<!-- user_preferences.html -->
{% load sitemessage %}

<form method="post">
    {% csrf_token %}

    <!-- Create preferences table from `subscr_prefs` template variable. -->
    {% sitemessage_prefs_table from subscr_prefs %}

    <input type="submit" value="Save preferences" />
</form>
```

Note: You can get subscribers as recipients list right from your message type, using *get_subscribers()* method.

3.8.3 Handling unsubscriptions

sitemessage bundles some views, and one of those allows users to unsubscribe from certain message types just by visiting it.

Please refer to *Bundled views* section of this documentation.

After that, for example, your E-mail client (if it supports *List-Unsubscribe* header) will happily introduce you some button to unsubscribe from messages of that type.

3.9 Bundled views

sitemessage bundles some views, and one of those allows users to unsubscribe from certain message types, or mark messages read just by visiting pages linked to those views. So let's configure your project to use those views:

```
from sitemessage.toolbox import get_sitemessage_urls

...

# Somewhere in your urls.py.

urlpatterns += get_sitemessage_urls()  # Attaching sitemessage URLs.
```

3.9.1 Unsubscribe

Read *Handling unsubscriptions* to get some information on how unsubscription works.

3.9.2 Mark read

When bundled views are attached to your app you can mark messages as read.

For example if you put the following code in your HTML e-mail message template, the message dispatch in your DB will be marked read as soon as Mail Client will render *<img>* tag.

```
{% if directive_mark_read %}
    <img src="{{ directive_mark_read }}">
{% endif %}
```

This allows to track whether a user has read a message.

3.10 Sitemessage for reusable applications

`sitemessage` offers reusable applications authors an API to send messages in a way that can be customized by project authors.

3.10.1 For applications authors

Use **sitemessage.toolbox.get_message_type_for_app** to return a registered message type object for your application.

Note: Project authors can override the above mentioned object to customize messages.

```
from sitemessage.toolbox import get_message_type_for_app, schedule_messages,␣
↪recipients


def schedule_email(text, to, subject):
    """Suppose you're sending a notification and want to sent a plain text e-mail by␣
↪default."""

    # This says: give me a message type `email_plain` if not overridden.
    message_cls = get_message_type_for_app('myapp', 'email_plain')
    message_obj = message_cls(subject, text)

    # And this actually schedules a message to send via `smtp` messenger.
    schedule_messages(message_obj, recipients('smtp', to))
```

Note: It's advisable for reusable applications authors to document which message types are used in the app by default, with which arguments, so that project authors may design their custom message classes accordingly.

3.10.2 For project authors

Use **sitemessage.toolbox.override_message_type_for_app** to override a given message type used by a certain application with a custom one.

Note: You'd probably need to know which message types are used in an app by default, and with which arguments, so that you may design your custom message classes accordingly (e.g. by subclassing the default type).

```
from sitemessage.toolbox import override_message_type_for_app

# This will override `email_plain` message type by `my_custom_email_plain` for␣
↪`myapp` application.
override_message_type_for_app('myapp', 'email_plain', 'my_custom_email_plain')
```

Warning: Be sure to call `override_message_type_for_app` beforehand. So that to the moment when a thirdparty app will try to send a message, message type is overridden.

CHAPTER 4

Get involved into django-sitemessage

Submit issues. If you spotted something weird in application behavior or want to propose a feature you can do that at https://github.com/idlesign/django-sitemessage/issues

Write code. If you are eager to participate in application development, fork it at https://github.com/idlesign/django-sitemessage, write your code, whether it should be a bugfix or a feature implementation, and make a pull request right from the forked project page.

Translate. If want to translate the application into your native language use Transifex: https://www.transifex.net/projects/p/django-sitemessage/.

Spread the word. If you have some tips and tricks or any other words in mind that you think might be of interest for the others — publish it.

CHAPTER 5

Also

If the application is not what you want for messaging with Django, you might be interested in considering other choices at https://www.djangopackages.com/grids/g/notification/ or https://www.djangopackages.com/grids/g/newsletter/ or https://www.djangopackages.com/grids/g/email/.

Python Module Index

Index