django-signal-notifier Documentation

Release 0.2.1

Mohammad Hadi Azaddel

Mar 21, 2021

CONTENTS

DSN or django-signal-notifier is a Django app to send message or notification based on the Django's signals triggering. You can assign some backends to each signal(e.g. An In-Site notification app).

The major difference between `django-signal-notifier` and other Django's notification packages is that *DSN* isn't just a simple message delivering system. It can act as a middleware between Django and every messenger client (Various clients like email, telegram, SMS and twitter).

It's working with event methodology, and it's based on Django signal. If a signal triggers, A messenger is called to send a message for specified users. To understand how it works, We explain some main concepts at first.

Attention: django-signal-notifier==0.2.1 is not compatible with **django>=3.1** . We are solving the problem.

CHAPTER

ONE

TABLE OF CONTENTS

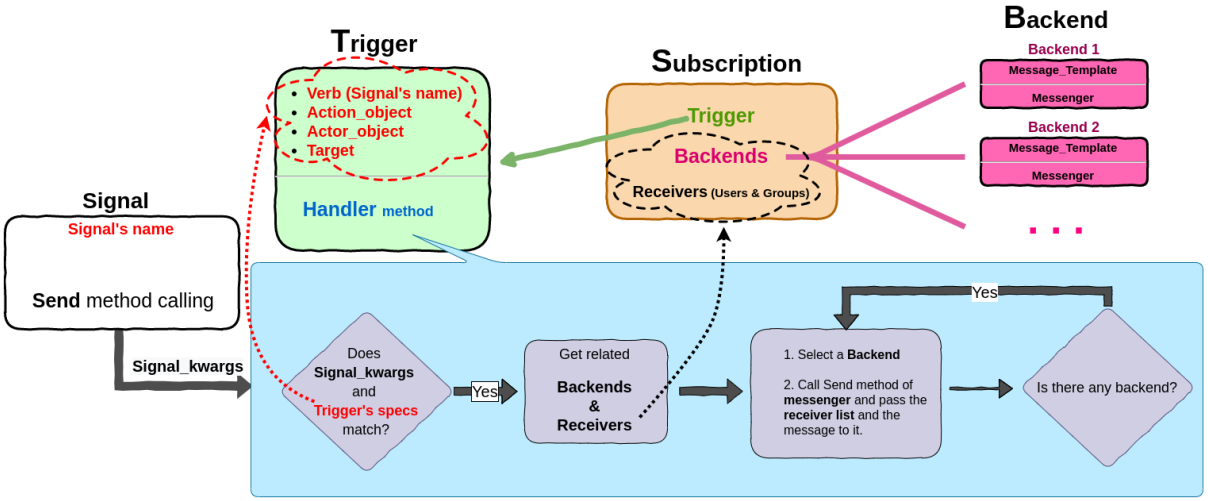
1.1 Quick Start

DSN or django-signal-notifier is a Django package to send message or notification based on the Django's signals triggering. You can select some backends for each signal(e.g., An In-Site notification app).

Attention:	django-signal-notifier==0.2.1 is not compatible with **django>=3.1** . We are solving the problem.

1.1.1 Concepts (Summary version)

DSN's architecture :



In a nutshell, we can say **DSN** is developed to *send message* :

- **When and Where** ? : When a Trigger Triggered (The associated signal's send function is called, and the trigger's specs match).
- **What** to send?: The message that is created to the message_template and other parameters like signal_kwargs.
- **Whom** to send? : Send the message to the registered receivers in the subscription or the dynamic user that can be specified in the messenger.

Note: You should pay attention to these 3 questions when you want to assign a new trigger to a signal.

1.1.2 Setup

Requirements

- Python 2.7, 3.4, 3.5, 3.6, 3.7
- Django 1.7, 1.8, 1.9, 1.10, 1.11, 2.0, 2.1, 2.2, 3.0

Attention: django-signal-notifier==0.2.1 is not compatible with **django>=3.1** . We are solving the problem.

Installation

1. Install `django-signal-notifier` by pip:

```
$ pip install django-signal-notifier
```

or use the source

```
$ git clone https://github.com/hadi2f244/django-signal-notifier
$ cd django-signal-notifier
$ python setup.py sdist
$ pip install dist/django-signal-notifier*
```

2. Add "django_signal_notifier" at the end of INSTALLED_APPS setting like this

```
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    ...
    'django_signal_notifier',
]
```

4. Migrate
5. `django-signal-notifier` configure by admin panel by default(Can be configured by code, tough)
6. Use `python manage.py migrate` for schema migration.

Attention:

You may face with below error, To resolve it, 'migrate' first.

```
no such table: django_signal_notifier_trigger.
An error occurs when reconnecting trigger to the corresponding signals, Note:␣
↪Make sure you migrate and migrations first
```

1.1.3 Usage

4. Run the development server and visit http://127.0.0.1:8000/admin/ To create a trigger(signal), backends(messenger and message_template), and subscription (you'll need the Admin app enabled).
5. **You can test it like this:** 5.1. Create a trigger (verb=pre_save and action_object=TestModel1)

   5.2. Create a backend (messenger=SimplePrintMessengerTemplateBased and message_template=SimplePrintMessageTemplate)

   5.3. Create a subscription that connects the trigger and the backend. Add admin to the receiver(user) list.

   5.4. Run this command in manage.py shell:

```
from django_signal_notifier.models import *
TestModel1_another_instance = TestModel1.objects.create(name="new_test_model2
↪", extra_field="extra")
```

   Now you should see a message when you create TestModel1. By Creating new TestModel1, Django calls the pre_save signal's send method. Then this signal call associated trigger handler. In the Trigger handler, the associated backend is called. The message_template with some details are sent to the backend. In our case, a simple message is printed. You can provide your messengers and message_templates.

1.2 Introduction

To understand how **DSN** works, We explain some main concepts at first.

1.2.1 Concepts

DSN has 3 main parts:

- **`Trigger`** Any Django's signal can be connected to the corresponding trigger. There is a one-to-one connection between each signal and each trigger.

  Calling the signal's **Send** method leads to calling the handler method of the corresponding trigger.

  Trigger has 4 parts(We got the idea from *Activity* concept in a similar package named Django-activity-stream):

  - `Verb`. The verb phrase that identifies the action of the activity.
  - `Action Object`. The object linked to the action itself.
  - `Actor Object`. *(Optional)* The object that performes the activity.
  - `Target`. *(Optional)* The object to which the activity was performed.

  Example: A telegram client can be defined as a Backend for **DSN**. We can define a trigger that is connected to `post_save` signal and one of the project models(*Testmodel*) set as the action_object. So when we create new *Testmodel*, the handler of the corresponding trigger is called automatically (The new *Testmodel* could be created by anyone and everywhere Because `actor_object` and `target` were left empty).

- **`Backend`** Backend is a tool to send a message like Notification, Email message, or So on that is the primary goal of **DSN**. We've got this idea from django-sitemessage): Backend consists of **two** parts:

  1. `Message_template`. It's a class as a template of the message that contains template string or points to a template file.

2. `Messenger`. It's the operational core of each backend that sends the string message (rendered message_template). E.g., Telegram and email Client.

Example: A telegram client can be defined as a Backend for **DSN**.

- **`Subscription`** Triggers and Backends are connected in a subscription entity. Message receivers are set in the subscription, too.

  If the handler method of a trigger is called, the related subscriptions receivers and backends are invoked. Then the backends(messenger) are called for each receiver(user). That's the central part of **DSN**.

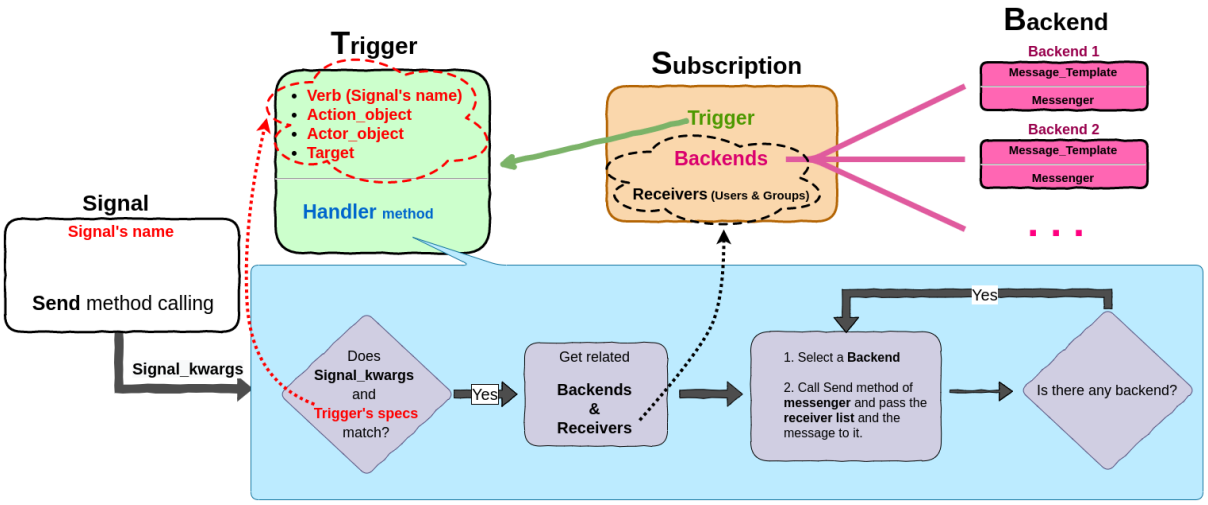
  Example: Same as the above example, A trigger that is connected to the post_save for *Testmodel* is defined. Also, we have connected a subscription that set email messenger as the backend and Admin user as the receiver. Therefore, If a new *Testmodel* object is created, an email message is sent to the Admin.

  Moreover, We can connect more than one subscription to a trigger. Subscriptions can be switched off.

Note: Receivers field are just provided in subscription for those situations that the receivers are static(e.g., sending some logs or notifications to Administrator user or group users). Besides, You can send the message to dynamic users(that changed according to the occasions). For more details refer to *Dynamic User* section.

1.2.2 Architecture

DSN's architecture :



As stated above, **DSN** consists of 3 models(Trigger, Subscription, and Backend). **DSN** works as follow:

1. **Setup and Initialization steps:** 1.1. Custom messengers, message_templates, and signals must be defined(*Optional*). It must be done through the code. 1.2. 3 steps must be done through the admin panel:

   1.2.1. Triggers must be defined by the name of the pre-defined signal(verb_name).

   1.2.2. Required backends must be defined by proper messenger and message_template.

   1.2.3. Subscriptions are the relations between the Trigger and Backends. So, according to the logic of our code, We must select proper backends for a trigger in a subscription.

2. **Execution: The code of DSN starts when a signal triggers(The send function calling).** 2.1. After the signal triggers, the handler method of the associated trigger is called, and It's check that passed signal arguments match the associated trigger.

   2.2. If everything matches, the associated subscription is evoked, Then a list of backends and receiver users are created.

   2.3. After that, each backend's messengers are called for the specified message and the user. (Note: We can set users dynamically. Hence associated user must be defined in the messenger, and the receiver field in the subscription must be left empty)

1.2.3 Summary

In a nutshell, we can say **DSN** is developed to *send message* :

- **When and Where** ? : When a Trigger Triggered (The associated signal's send function is called, and the trigger's specs match).
- **What** to send?: The message that is created to the message_template and other parameters like signal_kwargs.
- **Whom** to send? : Send the message to the registered receivers in the subscription or the dynamic user that can be specified in the messenger.

Note: It would be best if you took notice of these 3 questions When you want to assign a new trigger to a signal.

1.3 Setup

1.3.1 Requirements

- Python 2.7, 3.4, 3.5, 3.6, 3.7
- Django 1.7, 1.8, 1.9, 1.10, 1.11, 2.0, 2.1, 2.2, 3.0

Attention: django-signal-notifier==0.2.1 is not compatible with **django>=3.1** . We are solving the problem.

1.3.2 Installation

1. Install `django-signal-notifier` by pip:

```
$ pip install django-signal-notifier
```

or use the source

```
$ git clone https://github.com/hadi2f244/django-signal-notifier
$ cd django-signal-notifier
$ python setup.py sdist
$ pip install dist/django-signal-notifier*
```

2. Add "django_signal_notifier" at the end of INSTALLED_APPS setting like this

```
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    ...
    'django_signal_notifier',
]
```

3. `django-signal-notifier` configure by admin panel by default(Can be configured by code, tough)
4. Use `python manage.py migrate` for schema migration.

1.4 Usage

1.4.1 Initialization

As stated in Architecture *Architecture* part of *Introduction*, First You should implement a **trigger** regarding these parameters:

- `verb_name`: equals to the signal's name (*required*)
- `action_object`: An object or a class model that the signal operated on it. It is exactly equal to **sender** parameter in signals. (*It's required for the pre-defined Django signals*)
- `actor_object`: If you use a custom signal that you pass this parameter, you can use it (*Optional*)
- `target`: It's just a simple string and used as a piece of side information. Same as *actor_object*, *target* is used with a custom signal (*Optional*).

Then you can define some **backend** s. We have already defined some messengers and message_templates that are initialized on *DSN*'s core by default. You can implement your messengers and message_templates. It's explained in *Backends*.

For connecting backend to the trigger, you must define at least one **subscription** and select the user and group that must receive the message.

1.4.2 Custom Signal

In addition to django default signal(that we load all of them in **DSN** by default.), You can define your signal according to the Django official documentation. It's a standard way to define custom signals in the *signals.py* of each app.

```
custom_signal = Signal(providing_args=["parameter1"])
```

Then you must set up the custom signal in the *ready* function of the app's config in *apps.py*:

```
class MyAppConfig(AppConfig):
    name = 'myapp'
    ...
    def ready(self):
        from .signals import custom_signal
        from django_signal_notifier.models import Trigger
        ...
        Trigger.registered_verb_signal('custom_signal', custom_signal)
```

Attention: Because *apps.py* runs in migration too. To avoid initialization problems You **must** import django_signal_notifier and signals in ready function.

If you want to use **actor_object** or **target**, You must set them as the signal parameters. **action_object** is optional, but it isn't necessary to be defined as a parameter, You can set it as signal **sender** parameters(Refer to Django signal documentation, ` *send* method part <https://docs.djangoproject.com/en/3.0/topics/signals/#django.dispatch.Signal.send_robust>`_)

Note: **sender** is necessary for all pre-defined Django's signals. Therefore, *DSN* uses **sender** as **action_object** by default.

Same as a standard Django signal, you can use **send** and **send_robust** to trigger the signal.

1.5 Backends

All backend consists of two parts, messenger and message_template. You can select them in the admin panel, but if you need to define your messenger or message_template you must implement and add them to the messenger or message_template list of *DSN*.

1.5.1 Custom Message_template

Each message_template is a class which inherits from `BaseMessageTemplate` class(###link to the class##). We have to dissect `BaseMessageTemplate` and explain some details.

- **`file_name` and `template_string`:** Each message_template has a string template based on the Django template language. It can be a simple string, Html, etc. You can set it directly by *template_string* variable or set a file by *file_name*. At first, *DSN* checks *file_name* to get template string from it. Same as each Django app, template files are in **app_name/template/app_name**. So that, you must define that template file in the app that you defined new message_template class (You can refer to DSN_Notification ####links#### example for more details).

  ```
  file_name = "app_name/my_template.html"
  ```

  If *file_name* is left empty, *template_string* is set as the template string. There are no preferences between these two ways. Use whatever you prefer.

  An example:

  ```
  template_string = """
  {% if \"verb\" in context and context.verb != None %}
      <div>
          <p>{{ context.verb }}</p>
      </div>
  {% endif %} """
  ```

- **`render(self, user, trigger_context, signal_kwargs)`:** Messengers use this function to render message_template by the passed context. A Context is a dictionary which consists of three parts:

  - `user`: The User object that the message_template should render for that. We pass it to the message_template to access the user's name. (e.g., The user's name can be set at the message header).
  - `trigger_context`: It consists of *four trigger's parameters*.

- `signal_kwargs`: Other signal arguments that are passed to *DSN* can be accessed from this.

Note: You shouldn't change this function. We just explained this function to show how message_template class works. If you want to add more variables to the context, you should override the `get_template_context` function.

- **`get_template_context(self, context)`:** *User*, *trigger_context* and *signal_kwargs* are concatenated as *context*. You can set any new variables to *context*. For instance, you can add the sending time of a message to the template context:

```
def get_template_context(self, context):
    context['current_time'] = str(datetime.datetime.now().date())
    return context
```

Note: Notice that it doesn't call the superclass **get_template_context** method. So you should call parent's method manually in your code if you want:

```
def get_template_context(self, context):
    context = super().get_template_context(context)

    # Your code :
    ...
```

Briefly, You must set a template string or template_file for the message_template by `file_name` or `template_string`. To add more variables to the message context, You must overwrite `get_template_context` function.

1.5.2 Custom Messenger

Like message_template, every messenger is a class that inherits from a base class named `BaseMessenger` (###link to the class##). To define your messenger, You must redefine **send()** class method.

`send(self, template, sender, users, trigger_context, signal_kwargs):`

- `template`: This is the template object.
- `users`: List of users that you must send the message for them.

  Some messengers can send user's messages simultaneously to improve performance. Consequently, we avoid calling send function for each user singly. Instead, we left it to the messenger to send messages to users.
- `trigger_context`: Same as message_template
- `signal_kwargs`: Same as message_template

Firstly you must render the template class by `user`, `trigger_context`, `signal_kwargs`. You can render every user message by using a *for loop* over `users` list. Then you can send rendered string messages to the user. Example:

```
class simple_Messenger(BaseMessenger):
    @classmethod
    def send(self, template, users, trigger_context, signal_kwargs):
        for user in users:
            rendered_message = template.render(user=user, trigger_context=trigger_
↪context, signal_kwargs=signal_kwargs)
```

(continues on next page)

(continued from previous page)

```
My_messenger.send_my_message(user_receiver=user, context=rendered_message)
```

Note: For more details how to define a new message_template and messenger, refer to DSN_Notification ###link ##### documentation.

1.5.3 Add message_template and messenger

We suggest defining your messengers and message_templates in a separate file. E.g., messengers.py or message_template.py

You must introduce the new message_template and messenger to *DSN*. Use `Add_Messenger` and `Add_Message_Template` functions to add new messenger and message_template, respectively. You must do it in `ready()` function in *apps.py* of your app.

```
from django_signal_notifier.message_templates import Add_Message_Template
from django_signal_notifier.messengers import Add_Messenger

class MyAppConfig(AppConfig):
    ...

    def ready(self):
        from myapp.messengers import simple_Messenger
        from myapp.message_templates import simple_Message_template

        ...

        # Messengers :
        Add_Messenger(simple_Messenger)
        # Message templates :
        Add_Message_Template(simple_Message_template)
```

Attention: Because of that *apps.py* runs in migration. To avoid initialization problems You should import your messenger and message_template classes in `ready()` function.

After you re-run the app, you can see your messengers and message_templates are added to the messenger and message_template lists, respectively.

1.6 Dynamic User

You can choose some users and groups as the receiver of the message in the subscription model. Although there are many conditions that we want to set the receiver user dynamically. First, let's take a look at a scenario. Then we present the solution to the problem. Assume there are two models, `Movie` and `User`. We want to notify the audiences (``user``s) of a movie when it releases.

```
class Movie(models.Model):
    name = models.CharField(max_length=255)
    audiences = models.ManyToManyField(blank=True, to=User)
```

You probably defined a custom signal (e.g., *notify_audiences*) that you call it(send or send_rebust function) when you want to notify the audiences. So we don't discuss details of `Trigger` creation and the related process that occurs in **DSN** anymore. (Refer to *introduction*).

The custom signal can be

```
notify_audiences = Signal(providing_args=["movie"])
```

movie parameter is used to pass the movie object.

To specify the dynamic user, A messenger must be designed as follows:

```
class Notify_audiences_messenger(BaseMessenger):
    @classmethod
    def send(self, template, users, trigger_context, signal_kwargs):
        # Ignore the messenger when movie was not specified.
        # We did it to avoid calling this messenger by other asymmetric signals other␣
↪than notify_audiences
        try:
            movie = signal_kwargs['movie']
            audiences = movie.audiences
        except AttributeError:
            logger.error("Specified signal and Notify_audiences_messenger as backend␣
↪don't match together.")
            return

        for user in audiences:
            message = template.render(user=user, trigger_context=trigger_context,␣
↪signal_kwargs=signal_kwargs)

            # Send the rendered message to the user
            ...
```

As you see, the `users` argument(the set receivers in the subscription) is ignored, and a new user list is created. If you want to send the message to the preset receivers too, you can combine `users` and `audiences`.

Note: `user` parameter that template is rendered by, must be the type of `AUTH_USER_MODEL` (Refer to *settings*)

1.7 Background Tasks

1.8 Settings

- `AUTH_USER_MODEL` : `auth.User` is used as default user model. It must be a string in the format of *'app_name.user_model'*
- `PROFILE_MODEL` : According to the Django documentation, there are some ways to extend user model. But the simplest and extendable one is creating a user profile model that has a one-to-one connection to the 'auth.user' model. We defined a profile model as follow, You can change it in the format of *'app_name.user_model'*.

```
class DSN_Profile(models.Model):
    user = models.OneToOneField(to=app_settings.AUTH_USER_MODEL, on_delete=models.
↪CASCADE)
    telegram_chat_id = models.CharField(max_length=20, blank=True, null=True)
```

1.9 Comparison with similar projects

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search