As it exists tools to make his blog or website or even his own cloud system, “Trigger Happy” is a free software that provides a bridge to automatically share data between popular services you use on the web. And instead of giving your credentials to them, keep them with your own Trigger Happy to keep the control of your data!

1.1 Installation

To get the project do:

1.1.1 From GitHub

git clone https://github.com/foxmask/django-th.git

then continue by installing:

cd django-th
python setup install
cd ..
pip install -r requirements-evernote.txt

1.1.2 From Pypi

in 2 steps:

pip install django-th

and you will have to finish by

pip install -r https://raw.githubusercontent.com/foxmask/django-th/master/requirements-evernote.txt

this is because Evernote SDK for Python 3 is not yet available on pypi

1.1.3 Requirements

- Python 3.4.x
- Django >= 1.8
- arrow == 0.5.4
• django-formtools == 1.0
• django-js-reverse == 0.5.1
• libtidy-dev >= 0.99

The latest libtidy-dev should be installed with your operating system package manager, not from pip. On a Debian/Ubuntu system:

```
apt-get install libtidy-dev
```

for celery

• Celery == 3.1.18

for evernote support

• Evernote for python 3

for github support

• github == 1.0.0a2

for pocket support

• pocket == 0.3.5

for readability support

• readability == 1.0.0

for rss support

• feedparser == 5.1.3

for search engine

• django-haystack == 2.3.1

for trello support

• trello == 0.4.3
• py pandoc == 1.0.5

Pandoc is also needed of the system, that you can install on a Debian/Ubuntu system like this:

```
apt-get install pandoc
```

for twitter support

• twython == 3.2.0

for redis support

• django-redis == 4.1.0
• django-redisboard == 1.2.0

### 1.2 Configuration

Here are the details that will permit to make working the application correctly
1.2.1 urls.py

add this line to the urls.py to be able to use the complete application

url(r'', include('django_th.urls')),

this will give something like

```python
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'th.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),
    url(r'^admin/', include(admin.site.urls)),
    url(r'', include('django_th.urls')),
)
```

1.2.2 settings.py

add the module django_th, and its friends, to the INSTALLED_APPS

```python
INSTALLED_APPS = (
    ...
    'formtools',
    'django_js_reverse',
    'redisboard',
    'django_th',
    'th_rss',
    'th_pocket',
    'th_readability',
    'evernote',
    'th_evernote',
    'th_twitter',
    'th_holidays',
    'th_trello',
    'th_github',
    'haystack',  # mandatory if you plan to use th_search
    'th_search',  # then follow instructions from http://django-haystack.readthedocs.org/
    ...
)
```

this setting supposes you already own a Pocket account

**TH_SERVICES**

**TH_SERVICES** is a list of the services, like for example,

```python
TH_SERVICES = (  
    # comment the line to disable the service you dont want
    'th_rss.my_rss.ServiceRss',
    'th_pocket.my_pocket.ServicePocket',
    'th_evernote.my_evernote.ServiceEvernote',
    'th_readability.my_readability.ServiceReadability',
    'th_trello.my_trello.ServiceTrello',
)
```
1.2.3 Update the database

Once the settings is done, enter the following command to sync the database
if you start from scratch and don’t have created a django application yet, you should do :

```python
python manage.py syncdb
```
otherwise do :

```python
python manage.py migrate
```
if you meet some errors with this last command, have a look at MIGRATION_0.10.x_to_0.11.x.rst file

1.2.4 Activate the services

to activate a service, you will need to follow those steps

- Requesting a key to the Services
- Adding the key to your settings file
- Adding the service from the Admin
- Activating the service from your account from the public part of the website
- Why this process ?

**Requesting a key to the Services**

For each service, Trigger Happy expects to have some consumer key coming from the wanted service. So for each service, you need to register an account on each of this service, then required a key.

You can have a look at the README of Twitter, or README of Pocket

**Adding the key to the settings**

Once you own the keys., You add them to the settings.py file in

```python
TH_<SERVICE_NAME> = {
    'consumer_key' => 'foobar',
    'consumer_token' => 'blabla'
}
```

For example for Twitter :

```python
TH_TWITTER = {
    'consumer_key': 'abcdefghijklmnopqrstuvwxyz',
    'consumer_secret': 'abcdefghijklmnopqrstuvwxyz',
}
```
IMPORTANT:

With all the service you will enable, to avoid to share your key by accident, I strongly recommend that you put all of them in a separate local_settings.py that you include at the end of the main settings.py

So, when I speak about settings.py think about local_settings.py

Adding the service from the Admin

Once you did `python manage.py syncdb` and followed the standard process to bootstrap the application, go to the admin panel of the application.

Admin Home of Trigger Happy:

Admin list of activated services if Trigger Happy:

Admin Detail of one service of Trigger Happy:

Activating the service from your account from the public part of the website

Once your services are setup from the admin, you can go on the public part of the website and activate the service you need.

“My activated services”:

Why this process?

• it is simple: actually, to use Trigger Happy you need to install and host it by yourself, and so, you need to “declare” for each service your instance of TriggerHappy.

• Other details: you need to activate the service from the admin panel, BECAUSE, TriggerHappy is planned to be used by many other users soon. So the admin of the instance of TriggerHappy will decide if he wants to offer the possibility to use this service of this other one. Once the admin has done his job, the end user, from the “public part” can go to the list of services and add the new one etc.

1.2.5 Others settings

They are necessary if you want to be able to follow the log, cache rss and use the site framework

CACHE

For each TriggerHappy component, define one cache like below

```python
CACHES = {
    # Evernote Cache
    'th_evernote': {
        'TIMEOUT': 500,
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    },
    # Pocket Cache
```
{'th_pocket':
{
'TIMEOUT': 500,
"BACKEND": "django_redis.cache.RedisCache",
"LOCATION": "redis://127.0.0.1:6379/2",
"OPTIONS": {
    "CLIENT_CLASS": "django_redis.client.DefaultClient",
}
},
# RSS Cache
'th_rss':
{
'TIMEOUT': 500,
"BACKEND": "django_redis.cache.RedisCache",
"LOCATION": "redis://127.0.0.1:6379/3",
"OPTIONS": {
    "CLIENT_CLASS": "django_redis.client.DefaultClient",
}
},
# Readability
'th_readability':
{
'TIMEOUT': 500,
"BACKEND": "django_redis.cache.RedisCache",
"LOCATION": "redis://127.0.0.1:6379/4",
"OPTIONS": {
    "CLIENT_CLASS": "django_redis.client.DefaultClient",
}
},
# Trello Cache
'th_trello':
{
'TIMEOUT': 500,
"BACKEND": "django_redis.cache.RedisCache",
"LOCATION": "redis://127.0.0.1:6379/5",
"OPTIONS": {
    "CLIENT_CLASS": "django_redis.client.DefaultClient",
}
},
# Twitter Cache
'th_twitter':
{
'TIMEOUT': 500,
"BACKEND": "django_redis.cache.RedisCache",
"LOCATION": "redis://127.0.0.1:6379/6",
"OPTIONS": {
    "DB": 6,
    "CLIENT_CLASS": "django_redis.client.DefaultClient",
}
},
# Github Cache
'th_github':
{
'TIMEOUT': 500,
"BACKEND": "django_redis.cache.RedisCache",
"LOCATION": "redis://127.0.0.1:6379/7",
"OPTIONS": {
    "CLIENT_CLASS": "django_redis.client.DefaultClient",
}
The Log

The LOGGING adds to loggers

```
LOGGING = {
    'handlers': {
        'file': {
            'level': 'INFO',
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': BASE_DIR + '/trigger_happy.log',
            'maxBytes': 61280,
            'backupCount': 3,
            'formatter': 'verbose',
        },
    },
    'loggers': {
        'django_th.trigger_happy': {
            'handlers': ['console', 'file'],
            'level': 'INFO',
        },
    },
}
```

**CELELY**

Celery will handle tasks itself to populate the cache from provider services and then exploit it to publish the data to the expected consumer services

- From Settings

Define the broker then the scheduler

```
from celery.schedules import crontab

BROKER_URL = 'redis://localhost:6379/0'

CELERYBEAT_SCHEDULE = {
    'read-data': {
        'task': 'django_th.tasks.read_data',
        'schedule': crontab(minute='12,24,36,48'),
    },
    'publish-data': {
        'task': 'django_th.tasks.publish_data',
        'schedule': crontab(minute='20,40,59'),
    },
    'outside-cache': {
        'task': 'django_th.tasks.get_outside_cache',
        'schedule': crontab(minute='15,30,45'),
    },
}
```
An alternative configuration can be set here: If you prefer to enchain all the tasks in a raw, a dedicated task, named "go", will do the job so you will just need to do:

```python
CELERYBEAT_SCHEDULE = {
    'go': {
        'task': 'django_th.tasks.go',
        'schedule': crontab(minute='15,30,45'),
    },
}
```

that will replace the previous settings schedule

in your django application you will have to add 2 modules:

apps.py and celery.py:

in your app.py you will need to add:

```python
from .celery import app as celery_app
```

and in the celery.py:

```python
from __future__ import absolute_import
import os
from celery import Celery
from django.conf import settings

# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', '<YOUR_DJANGO_APP_NAME>.settings')

app = Celery('<YOUR_DJANGO_APP_NAME>')

# Using a string here means the worker will not have to
# pickle the object when using Windows.
app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

that way, Celery will read your settings instead of the one provided by django_th too.

**SUPERVISORD**

```
[program:django_th_worker]
user = foxmask
directory=/home/projects/trigger-happy/th
command=/home/projects/trigger-happy/bin/celery -A django_th worker --autoscale=10,3 -l info
autostart=true
autorestart=true
redirect_stderr=true
stdout_logfile=/home/projects/trigger-happy/logs/trigger-happy.log
stderr_logfile=/home/projects/trigger-happy/logs/trigger-happy-err.log

[program:django_th_beat]
user = foxmask
directory=/home/projects/trigger-happy/th
command=/home/projects/trigger-happy/bin/celery -A django_th beat -l info
autostart=true
```
autorestart=true
redirect_stderr=true
stdout_logfile=/home/projects/trigger-happy/logs/trigger-happy.log
stderr_logfile=/home/projects/trigger-happy/logs/trigger-happy-err.log

**REDISBOARD**

```python
# REDISBOARD
REDISBOARD_DETAIL_FILTERS = ['.*']
```

**HAYSTACK**

if you plan to use the search feature, put the engine of your choice, for example:

```python
# needed to th_search and haystack
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',
        'URL': 'http://127.0.0.1:9200/',
        'INDEX_NAME': 'haystack',
    },
}
```

### 1.3 Usage

#### 1.3.1 Activating services:

The user activates the service for their own need. If the service requires an external authentication, he will be redirected to the service which will ask him the authorization to access the user’s account. Once it’s done, goes back to django-trigger-happy to finish and record the “auth token”.

#### 1.3.2 Using the activated services:

A set of 3 pages will ask to the user information that will permit to trigger data from a service “provider” to a service “consumer”.

For example:

- page 1: the user gives a RSS feed
- page 2: the user gives the name of the notebook where notes will be stored and a tag if he wants
- page 3: the user gives a description

**Fire the Triggers**:

To start handling the queue of triggers you/your users configure, just set the `fire.py` in a crontab or any other scheduler solution of your choice. Keep in mind to avoid to set a too short duration between 2 run to avoid to be blocked by the externals services you/your users want to reach.
1.4 Create a new module

1.4.1 Introduction:

You can start a new module by cloning the project Django Th Dummy which is a vanilla django module, ready to be used, after you’ve replaced the name of the form/model/class we’ll see below.

Once you’ve cloned it, rename the folder th_dummy to the name of your choice.

Below we’ll keep the name dummy to continue our explanation.

1.4.2 Forms:

The form th_dummy/forms.py provides 3 forms:

- DummyForm a modelForm
- DummyFormProvider which extends DummyForm
- DummyFormConsumer which extends DummyForm

DummyForm will define the content of our form, our fields our widget etc.

1.4.3 Models:

The model th_dummy/models.py:

```python
class Dummy(Services):
    # put whatever you need here
    # eg title = models.CharField(max_length=80)
    # but keep at least this one
    title = models.CharField(max_length=80)
    trigger = models.ForeignKey('TriggerService')

    class Meta:
        app_label = 'django_th'

    def __str__(self):
        return "%s" % (self.name)

    def show(self):
        return "My Dummy %s" % (self.name)
```

Key points:

- The model is related to TriggerService model
- The model uses the app_label to django_th meta, so the Trigger Happy will be added the table name

1.4.4 Service class:

At the beginning of the class ServiceDummy (from th_dummy/my_dummy.py) you will need to import the class of the third party application.
the class ServiceDummy will extend ServiceMgr we’ve imported from django_th.services.services This class is composed at least by 2 methods :

process_data :

we provide the following parms

- token - the token of the service
- trigger_id - the trigger id we handle
- date_triggered - the date of the last trigger

role : grabs the data of the current service to be provided to another

return : a list composed by : title, url, content, and can return also my_date a datetime value

save_data :

we provider the following parms

- token - the token of the service
- trigger_id - the trigger id we handle,
- data - the data to store (title, url, content), provided by a “process_data” of another service

role : save the data to the ServiceDummy

return : a boolean True or False, if the save_data worked fine or not

If the service does not save data, it’s the case of the module django-th-rss which just provides stuff and save nothing, you’ll put pass to save_data as the body of your code

auth and callback :

If your service need an authentication, you’ll need 2 new functions auth and callback

- auth will trigger the authentication to the third party application, the Oauth process in fact
- callback is triggered when the authentication is done and call by the third party application.

At this step the callback function store the oauth token to the dedicated dummy model

The complete code of this class :

```python
# coding: utf-8
# add here the call of any native lib of python like datetime etc.
#
# add the python API here if needed
from external_api import CallOfApi

# django classes
from django.conf import settings
from django.utils.log import getLogger

# django_th classes
from django_th.services.services import ServicesMgr
```

1.4. Create a new module
from django_th.models import UserService, ServicesActivated

""
handle process with dummy
put the following in settings.py

TH_DUMMY = {
    'consumer_key': 'abcdefghijklmnopqrstuvwxyz',
}

TH_SERVICES = (
    ...
    'th_dummy.my_dummy.ServiceDummy',
    ...
)
""

logger = getLogger('django_th.trigger_happy')

class ServiceDummy(ServicesMgr):

    def __init__(self):
        self.dummy_instance = external_api.CallOfApi(
            settings.TH_DUMMY['consumer_key'], token)

    def read_data(self, token, trigger_id, date_triggered):
        ""
        get the data from the service
        :param trigger_id: trigger ID to process
        :param date_triggered: the date of the last trigger
        :type trigger_id: int
        :type date_triggered: datetime
        :return: list of data found from the date_triggered filter
        :rtype: list
        ""
        data = list()
        return cache.set('th_dummy_' + str(trigger_id), data)

    def process_data(self, trigger_id):
        ""
        get the data from the cache
        :param trigger_id: trigger ID from which to save data
        :type trigger_id: int
        ""
        return super(ServiceDummy, self).process_data('th_dummy', str(trigger_id))

    def save_data(self, token, trigger_id, **data):
        ""
        let's save the data
        :param trigger_id: trigger ID from which to save data
        :param **data: the data to check to be used and save
        :type trigger_id: int
        :type **data: dict
        :return: the status of the save statement
        ""
:rtype: boolean

```
from th_dummy.models import Dummy
status = False

if token and 'link' in data and data['link'] is not None and len(data['link']) > 0:
    # get the data of this trigger
    trigger = Dummy.objects.get(trigger_id=trigger_id)
    # if the external service need we provide
    # our stored token and token secret then I do
    # token_key, token_secret = token.split('#TH#')

    title = ''
    title = (data['title'] if 'title' in data else '')
    # add data to the external service
    item_id = self.dummy_instance.add(
        url=data['link'], title=title, tags=(trigger.tag.lower()))

    sentence = str('dummy {} created').format(data['link'])
    logger.debug(sentence)
    status = True

else:
    logger.critical('no token or link provided for trigger ID {}').format(trigger_id)
    status = False

return status

def auth(self, request):
    
    let's auth the user to the Service
    
    request_token = super(ServiceDummy, self).auth(request)
    callback_url = self.callback_url(request, 'dummy')

    # URL to redirect user to, to authorize your app
    auth_url_str = '%s?oauth_token=%s&oauth_callback=%s'
    auth_url = auth_url_str % (self.AUTH_URL, request_token['oauth_token'], callback_url)

    return auth_url

def callback(self, request):
    
    Called from the Service when the user accept to activate it
    
    kwargs = {'access_token': '', 'service': 'ServiceDummy',
              'return': 'dummy'}
    return super(ServiceDummy, self).callback(request, **kwargs)
```

1.5 MIGRATIONS from 0.10.x to 0.11.x:

Nota: in the SQL queries below, I use CURRENT_TIMESTAMP because of Postgresql. Adapt it to your own RDBMS.
1.5.1 Django Trigger Happy tables:

To migrate enter,

```
python manage.py migrate
```

if the migration complains that you’ve already created the table django_th_rss then check the follow:

```
select * from django_migrations ;
```

to find

```
11 | django_th | 0001_initial | 2015-06-10 10:00:00.977958+02
```

if you don’t have it then do:

```
insert into django_migrations (app,name,applied) values('django_th','0001_initial',CURRENT_TIMESTAMP);
```

then replay

```
python manage.py migrate
```

1.5.2 Django Trigger Happy Module tables:

**Evernote**:

if the migration complains that you’ve already created the table django_th_evernote then check it by:

```
select * from django_migrations ;
```

check that you don’t have those record in the django_migrations table

```
13 | th_evernote | 0001_initial | 2015-06-10 10:00:00.977958+02
```

if it’s not the case, then add the following by hand like that:

```
insert into django_migrations (app,name,applied) values('th_evernote','0001_initial',CURRENT_TIMESTAMP);
```

**Holidays**:

if the migration complains that you’ve already created the table django_th_holidays then check it by:

```
select * from django_migrations ;
```

check that you don’t have those record in the django_migrations table

```
13 | th_holidays | 0001_initial | 2015-06-10 10:00:00.977958+02
```

if it’s not the case, then add the following by hand like that:

```
insert into django_migrations (app,name,applied) values('th_holidays','0001_initial',CURRENT_TIMESTAMP);
```
Pocket:

if the migration complains that you’ve already created the table django_th_pocket then check it by:

```sql
select * from django_migrations;
```

check that you don’t have those record in the django_migrations table

```sql
select * from django_migrations;
```

| 13 | th_pocket | 0001_initial | 2015-06-10 10:00:00.977958+02 |

if it’s not the case, then add the following by hand like that:

```sql
insert into django_migrations (app,name,applied) values('th_pocket','0001_initial',CURRENT_TIMESTAMP);
```

Readability:

if the migration complains that you’ve already created the table django_th_readability then check it by:

```sql
select * from django_migrations;
```

check that you don’t have those record in the django_migrations table

```sql
select * from django_migrations;
```

| 13 | th_readability | 0001_initial | 2015-06-10 10:00:00.977958+02 |

if it’s not the case, then add the following by hand like that:

```sql
insert into django_migrations (app,name,applied) values('th_readability','0001_initial',CURRENT_TIMESTAMP);
```

Twitter:

if the migration complains that you’ve already created the table django_th_twitter then check it by:

```sql
select * from django_migrations;
```

check that you don’t have those record in the django_migrations table

```sql
select * from django_migrations;
```

| 13 | th_twitter | 0001_initial | 2015-06-10 10:00:00.977958+02 |

if it’s not the case, then add the following by hand like that:

```sql
insert into django_migrations (app,name,applied) values('th_twitter','0001_initial',CURRENT_TIMESTAMP);
```

```sql
insert into django_migrations (app,name,applied) values('th_twitter','0002_int_to_bigint',CURRENT_TIMESTAMP);
```

before adding by hand the line below, check that the table django_th_twitter contains the column max_id and since_id as bigint and not just int

if that columns are not bigint add just this

```sql
insert into django_migrations (app,name,applied) values('th_twitter','0001_initial',CURRENT_TIMESTAMP);
```

otherwise add this too

1.5. MIGRATIONS from 0.10.x to 0.11.x :
insert into django_migrations (app, name, applied) values('th_twitter', '0002_int_to_bigint', CURRENT_TIMESTAMP);

Table to drop:

with the last

python manage.py migrate

you will meet this message:

Running migrations:
   No migrations to apply.
   Your models have changes that are not yet reflected in a migration, and so won't be applied.
   Run 'manage.py makemigrations' to make new migrations, and then re-run 'manage.py migrate' to apply them.
The following content types are stale and need to be deleted:

    django_th | userprofile

answer yes as this one is not used at all

then play again

python manage.py migrate

thus the migration will skip that steps and will continue smoothly

For example a new RSS item is published, Trigger Happy will be able to automatically create a note on your Evernote account or create a bookmark to your own Readability or Pocket account and so on
The goal of this project is to be independent from any other solution like IFTTT, CloudWork or others. Thus you could host your own solution and manage your own triggers without depending any non-free solution. With this project you can host triggers for you. All you need is to have a hosting provider (or simply your own server ;) who permits to use a manager of tasks like “cron” and, of course Python.
You can find all details of all existing services of the blog
Indices and tables

• genindex
• modindex
• search