django-notifications-hq 1.6.0

Latest version

Released: Feb 22, 2020

```
pip install django-notifications-hq
```

GitHub notifications alike app for Django.

Navigation

Project description

Release history

Download files

Project links

Homepage

Statistics

Project description

`django-notifications` Documentation

django-notifications is a GitHub notification alike app for Django, it was derived from django-activity-stream

The major difference between `django-notifications` and `django-activity-stream`:

- `django-notifications` is for building something like Github "Notifications"
- While `django-activity-stream` is for building Github "News Feed"

GitHub statistics:

- **Stars:** 1,048
- **Forks:** 331
- **Open issues/PRs:** 52

View statistics for this project via Libraries.io, or by using our public dataset on Google BigQuery

Meta

License: BSD License (MIT)

Author: django-notifications team

django, notifications, github, action, event, stream

Maintainers

alvaro.lqueiroz

brantyoung

nemesisdesign

Classifiers

Notifications are actually actions events, which are categorized by four main components.

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

`Actor`, `Action Object` and `Target` are `GenericForeignKeys` to any arbitrary Django object. An action is a description of an action that was performed (`Verb`) at some instant in time by some `Actor` on some optional `Target` that results in an `Action Object` getting created/updated/deleted.

For example: justquick `(actor)` *closed* `(verb)` issue 2 `(action_object)` on activity-stream `(target)` 12 hours ago

Nomenclature of this specification is based on the Activity Streams Spec: http://activitystrea.ms/specs/atom/1.0/

Requirements

- Python 3.5, 3.6, 3.7, 3.8
- Django 2.2, 3.0

Installation

Installation is easy using `pip` and will install all required libraries.

```
$ pip install django-notifications-hq
```

or get it from source

```
$ git clone https://github.com/django-notifications/djan
$ cd django-notifications
$ python setup.py sdist
$ pip install dist/django-notifications-hq*
```

Development Status
- 5 - Production/Stable

Environment
- Web Environment

Framework
- Django
- Django :: 2.2
- Django :: 3.0

Intended Audience
- Developers

License
- OSI Approved :: BSD License

Operating System
- OS Independent

Programming Language
- Python
- Python :: 3
- Python :: 3.5
- Python :: 3.6
- Python :: 3.7
- Python :: 3.8

Topic
- Utilities

Note that django-model-utils will be installed: this is required for the pass-through QuerySet manager.

Then to add the Django Notifications to your project add the app `notifications` to your `INSTALLED_APPS` and urlconf.

The app should go somewhere after all the apps that are going to be generating notifications like `django.contrib.auth`

```
INSTALLED_APPS = (
    'django.contrib.auth',
    ...
    'notifications',
    ...
)
```

Add the notifications urls to your urlconf:

```
import notifications.urls

urlpatterns = [
    ...
    url('^inbox/notifications/', include(notifications.u
    ...
]
```

The method of installing these urls, importing rather than using `'notifications.urls'`, is required to ensure that the urls are installed in the `notifications` namespace.

To run schema migration, execute `python manage.py migrate notifications`.

Generating Notifications

Generating notifications is probably best done in a separate signal.

```
from django.db.models.signals import post_save
from notifications.signals import notify
from myapp.models import MyModel

def my_handler(sender, instance, created, **kwargs):
    notify.send(instance, verb='was saved')

post_save.connect(my_handler, sender=MyModel)
```

To generate an notification anywhere in your code, simply import the notify signal and send it with your actor, recipient, and verb.

```
from notifications.signals import notify

notify.send(user, recipient=user, verb='you reached leve
```

The complete syntax is.

```
notify.send(actor, recipient, verb, action_object, targe
```

Arguments:

- **actor**: An object of any type. (Required) Note: Use **sender** instead of **actor** if you intend to use keyword arguments
- **recipient**: A **Group** or a **User QuerySet** or a list of **User**. (Required)
- **verb**: An string. (Required)
- **action_object**: An object of any type. (Optional)
- **target**: An object of any type. (Optional)
- **level**: One of Notification.LEVELS ('success', 'info', 'warning', 'error') (default=info). (Optional)
- **description**: An string. (Optional)
- **public**: An boolean (default=True). (Optional)
- **timestamp**: An tzinfo (default=timezone.now()). (Optional)

Extra data

You can attach arbitrary data to your notifications by doing the following:

- Add to your settings.py: `DJANGO_NOTIFICATIONS_CONFIG = { 'USE_JSONFIELD': True}`

Then, any extra arguments you pass to `notify.send(...)` will be attached to the `.data` attribute of the notification object. These will be serialised using the JSONField's serialiser, so you may need to take that into account: using only objects that will be serialised is a good idea.

Soft delete

By default, `delete/(?P<slug>\d+)/` deletes specified notification record from DB. You can change this behaviour to "mark `Notification.deleted` field as `True`" by:

- Add to your settings.py: `DJANGO_NOTIFICATIONS_CONFIG = { 'SOFT_DELETE': True}`

With this option, QuerySet methods `unread` and `read` contain one more filter: `deleted=False`. Meanwhile, QuerySet methods `deleted`, `active`, `mark_all_as_deleted`, `mark_all_as_active` are turned on. See more details in QuerySet methods section.

API

QuerySet methods

Using `django-model-utils`, we get the ability to add queryset methods to not only the manager, but to all querysets that will be used, including related objects. This enables us to do things like:

```
Notification.objects.unread()
```

which returns all unread notifications. To do this for a single user, we can do:

```
user = User.objects.get(pk=pk)
user.notifications.unread()
```

There are some other QuerySet methods, too.

`qs.unsent()`

Return all of the unsent notifications, filtering the current queryset. (emailed=False)

`qs.sent()`

Return all of the sent notifications, filtering the current queryset. (emailed=True)

`qs.unread()`

Return all of the unread notifications, filtering the current queryset. When `SOFT_DELETE=True`, this filter contains `deleted=False`.

`qs.read()`

Return all of the read notifications, filtering the current queryset. When `SOFT_DELETE=True`, this filter contains `deleted=False`.

`qs.mark_all_as_read()` | `qs.mark_all_as_read(recipient)`

Mark all of the unread notifications in the queryset (optionally also filtered by `recipient`) as read.

`qs.mark_all_as_unread()` | `qs.mark_all_as_unread(recipient)`

Mark all of the read notifications in the queryset (optionally also filtered by `recipient`) as unread.

`qs.mark_as_sent()` | `qs.mark_as_sent(recipient)`

Mark all of the unsent notifications in the queryset (optionally also filtered by `recipient`) as sent.

`qs.mark_as_unsent()` | `qs.mark_as_unsent(recipient)`

Mark all of the sent notifications in the queryset (optionally also filtered by `recipient`) as unsent.

`qs.deleted()`

Return all notifications that have `deleted=True`, filtering the current queryset. Must be used with `SOFT_DELETE=True`.

`qs.active()`

Return all notifications that have `deleted=False`, filtering the current queryset. Must be used with `DELETE=True`.

`qs.mark_all_as_deleted()` | `qs.mark_all_as_deleted(recipient)`

Mark all notifications in the queryset (optionally also filtered by `recipient`) as `deleted=True`. Must be used with `DELETE=True`.

`qs.mark_all_as_active()` | `qs.mark_all_as_active(recipient)`

Mark all notifications in the queryset (optionally also filtered by `recipient`) as `deleted=False`. Must be used with `SOFT_DELETE=True`.

Model methods

`obj.timesince([datetime])`

A wrapper for Django's `timesince` function.

`obj.mark_as_read()`

Mark the current object as read.

Template tags

Put *{% load notifications_tags %}* in the template before you actually use notification tags.

`notifications_unread`

```
{% notifications_unread %}
```

Give the number of unread notifications for a user, or nothing (an empty string) for an anonymous user.

Storing the count in a variable for further processing is advised, such as:

```
{% notifications_unread as unread_count %}
...
{% if unread_count %}
    You have <strong>{{ unread_count }}</strong> unread
{% endif %}
```

Live-updater API

To ensure users always have the most up-to-date notifications, *django-notifications* includes a simple javascript API for updating specific fields within a django template.

There are two possible API calls that can be made:

1. `api/unread_count/` that returns a javascript object with 1 key: `unread_count` eg:

```
{"unread_count":1}
```

2. `api/unread_list/` that returns a javascript object with 2 keys: *unread_count* and *unread_list* eg:

```
{
 "unread_count":1,
 "unread_list":[--list of json representations of not
}
```

Representations of notifications are based on the django method: `model_to_dict`

Query string arguments:

- **max -** maximum length of unread list.
- **mark_as_read -** mark notification in list as read.

For example, get `api/unread_list/?max=3&mark_as_read=true` returns 3 notifications and mark them read (remove from list on next request).

How to use:

1. Put `{% load notifications_tags %}` in the template before you actually use notification tags.
2. In the area where you are loading javascript resources add the following tags in the order below:

```
<script src="{% static 'notifications/notify.js' %}"
{% register_notify_callbacks callbacks='fill_notifica
```

`register_notify_callbacks` takes the following arguments:

1. `badge_class` (default `live_notify_badge`) - The identifier *class* of the element to show the unread count, that will be

periodically updated.

2. `menu_class` (default `live_notify_list`) - The identifier *class* of the element to insert a list of unread items, that will be periodically updated.
3. `refresh_period` (default `15`) - How often to fetch unread items from the server (integer in seconds).
4. `fetch` (default `5`) - How many notifications to fetch each time.
5. `callbacks` (default `<empty string>`) - A comma-separated list of javascript functions to call each period.
6. `api_name` (default `list`) - The name of the API to call (this can be either `list` or `count`).

3. To insert a live-updating unread count, use the following template:

```
{% live_notify_badge %}
```

`live_notify_badge` takes the following arguments:

1. `badge_class` (default `live_notify_badge`) - The identifier `class` for the `<span>` element that will be created to show the unread count.

4. To insert a live-updating unread list, use the following template:

```
{% live_notify_list %}
```

`live_notify_list` takes the following arguments:

1. `list_class` (default `live_notify_list`) - The identifier `class` for the `<ul>` element that will be created to insert the list of notifications into.

Using the live-updater with bootstrap

The Live-updater can be incorporated into bootstrap with minimal code.

To create a live-updating bootstrap badge containing the unread count, simply use the template tag:

```
{% live_notify_badge badge_class="badge" %}
```

To create a live-updating bootstrap dropdown menu containing a selection of recent unread notifications, simply use the template tag:

```
{% live_notify_list list_class="dropdown-menu" %}
```

Customising the display of notifications using javascript callbacks

While the live notifier for unread counts should suit most use cases, users may wish to alter how unread notifications are shown.

The `callbacks` argument of the `register_notify_callbacks` dictates which javascript functions are called when the unread api call is made.

To add a custom javascript callback, simply add this to the list, like so:

```
{% register_notify_callbacks callbacks='fill_notificatio
```

The above would cause the callback to update the unread count badge, and would call the custom function *my_special_notification_callback*. All callback functions are passed a single argument by convention called *data*, which contains the entire result from the API.

For example, the below function would get the recent list of unread messages and log them to the console:

```
function my_special_notification_callback(data) {
    for (var i=0; i < data.unread_list.length; i++) {
        msg = data.unread_list[i];
        console.log(msg);
    }
}
```

Testing the live-updater

1. Clone the repo
2. Run *./manage.py runserver*
3. Browse to *yourserverip/test/*
4. Click 'Make a notification' and a new notification should appear in the list in 5-10 seconds.

Serializing the django-notifications Model

See here - http://www.django-rest-framework.org/api-guide/relations/#generic-relationships

In this example the target object can be of type Foo or Bar and the appropriate serializer will be used.

```
class GenericNotificationRelatedField(serializers.Relate

    def to_representation(self, value):
        if isinstance(value, Foo):
            serializer = FooSerializer(value)
        if isinstance(value, Bar):
            serializer = BarSerializer(value)

        return serializer.data


class NotificationSerializer(serializers.Serializer):
    recipient = PublicUserSerializer(User, read_only=Tru
    unread = serializers.BooleanField(read_only=True)
    target = GenericNotificationRelatedField(read_only=T
```

Thanks to @DaWy

`AbstractNotification` model

In case you need to customize the notification model in order to add field or customised features that depend on your application, you can inherit and extend the `AbstractNotification` model, example:

```
from django.db import models
from notifications.base.models import AbstractNotificati


class Notification(AbstractNotification):
    # custom field example
    category = models.ForeignKey('myapp.Category',
                                 on_delete=models.CASCAD

    class Meta(AbstractNotification.Meta):
        abstract = False
```

Notes

Email Notification

Sending email to users has not been integrated into this library. So for now you need to implement it if needed. There is a reserved field *Notification.emailed* to make it easier.

`django-notifications` Team

Core contributors (in alphabetical order):

- Alvaro Leonel
- Samuel Spencer
- Yang Yubo
- Zhongyuan Zhang

Help About PyPI

Installing packages
Uploading packages
User guide
FAQs

PyPI on Twitter
Infrastructure dashboard
Package index name retention
Our sponsors

Contributing to PyPI

Bugs and feedback
Contribute on GitHub
Translate PyPI
Development credits

Using PyPI

Code of conduct
Report security issue
Privacy policy
Terms of use

Status: Partially Degraded Service

Developed and maintained by the Python community, for the Python community.
Donate today!

Site map

Switch to desktop version

English español français 日本語 português (Brasil) українська Ελληνικά Deutsch 中文 (简体) русский עברית

Pingdom
Monitoring

Google
Object Storage and Download Analytics

Sentry
Error logging

AWS
Cloud computing

DataDog
Monitoring

Fastly
CDN

DigiCert
EV certificate

StatusPage
Status page