Django Notify Documentation

Release 0.1

Vikas Yadav

Apr 24, 2018

Contents

CHAPTER 1

Contents:

1.1 Introduction

1.1.1 About the app

A normal web app with feature of social interaction requires to keep their users updated about recent happenings. For this, a notification system comes handy which stores numerous kinds of notifications and updates the user that a notification is received as soon as an action is performed.

The sole purpose of `django-notify-x` is to add a *facebook-like* notification system functionality and give you the ability to control everything.

The key features of Django Notify are:

- Send notifications to multiple user at once.
- AJAX supported views to update, read and delete notifications on the client-side.
- To provide activity-stream like field options.
- To distinguish notifications by their types, so that you can control their behaviours easily.
- To handle client-side javascript, the django way.
- And many more!

1.1.2 Disclaimer

This project may seem like a replica of django-notifications , yes, this project was highly inspired from `django-notifications`.

But to make it clear, when I started this, too many features were already commited on the project and they used different approach from what I wanted. So instead of forking, doing too many changes and sending a PR, I started my own version of it :).

1.2 Getting Started

1.2.1 Installation

Install a stable from PyPI using `pip`:

```
$ pip install django-notify-x
```

Install latest commit from Github:

```
$ pip install -e git+git://github.com/v1k45/django-notify-x.git#egg=notify
```

Add `notify` in `INSTALLED_APPS` of your project settings:

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

Include `notify.urls` in your `urls.py` with `notifications` as namespace:

```
urlpatterns = [
    ...
    url(r'^notifications/', include('notify.urls', 'notifications')),
    ...
]
```

Finally, run migrations:

```
$ python manage.py migrate notify
```

1.2.2 Dependencies

`django-notify-x` currently supports Django 1.8 and above. There is no support for previous versions. Both, Python 2.7 as well as Python 3.4 are supported.

It uses `django-jsonfield` package to add support to attach JSON data to notifications using the `extra` field.

1.3 Using Django Notify

This page will describe how to use `django-notify-x`.

1.3.1 Component details

Just like `django-notifications`, this app also follows the approach described by ActivityStrea.ms . Notifications are generated when an action is performed on concerning a recipient.

The following are the main components:

- **Actor**: The object which performed the activity.
- **Verb**: The activity.

- **Object**: The object on which activity was performed.
- **Target**: The object where activity was performed.

These parameters are nothing but `GenericForeignKey` relation to an arbitrary Django model object.

The parameters **Action Object** and **Target** can be left optional. Even **Actor** in somecases can be said as optional, there can be many cases for this. Therefore, all parameters can have *Anonymous* objects, not objects actually, but texts.

All three parameters, *Actor, Action Object and, Target* have their respective fields where the object property can be left empty and a simple text value can be used instead.

1.3.2 Example notification cases

- **John followed you. XX minutes ago.**
  - `<actor> <verb> <created>`
- **Jane commented on your post *Django is fun*. XX minutes ago.**
  - `<actor> <verb> <target> <created>`
- **John replied on your *comment* on *Django is fun*. XX minutes ago.**
  - `<actor> <verb> <object> <target> <created>`
- **You received 20 points. Today.**
  - `<actor_text> <verb> <created>`
- *and many more...*

1.3.3 Sending Notifications

Sending notifications to single user

```
from notify.signals import notify

# your example view
def follow_user(request, user):
    user = User.objects.get(username=user)
    ...
    dofollow
    ...

    notify.send(request.user, recipient=user, actor=request.user
                verb='followed you.', nf_type='followed_by_one_user')

    return YourResponse
```

Easy as pie, isn't it?

Sending notifications to to multiple users:

```
from notify.signals import notify

# your example view
def upload_video(request):
```

```
    ...
    uploadvideo...
    ...
    video = VideoUploader.getupload()
    followers = list(request.user.followers())

    notify.send(request.user, recipient_list=followers, actor=request.user
                verb='uploaded.', target=video, nf_type='video_upload_from_following')

    return YourResponse
```

Just change the `recipient` to `recipient_list` and send notifications to as many users you want!

Warning: `recipient_list` expects supplied object to be a `list()` or `set()` instance, make sure you convert your `QuerySet` to `set()` or `list()` before assigning value. `set()` is preferred because it eliminates duplicate values and membership checks are faster.

1.4 Further configurations

After you've installed the app and made an entry in the `INSTALLED_APPS` of your project settings, there are things that you still need to know about.

If you are planning to handle things over AJAX, make sure you run this command before starting to hack the app:

```
$ python manage.py collectstatic
```

This will copy `notifyX.js` and `notifyX.min.js` to your static root directory.

The `notifyX.js` file handles all the AJAX related actions ranging from marking notifications as read/unread to fetching new notifications and updating the notification box.

Now, include the `notifyX.js` static file in your base template.:

```
{% load staticfiles %}
....
<script src="{% static "notify/notifyX.js" %}"></script>
<!-- OR -->
<script src="{% static "notify/notifyX.min.js" %}"></script>
```

Note: The other file, `notifyX.min.js` is a minified version of `notifyX.js`. It is intended work the same as the original file. The advantage of a minified javascript file is that it removes unnecessary things and ultimately reduces the size of the final file with same functionality.

Warning: Make sure you include `notifyX.js` after you've included latest JQuery javascript file. This script uses JQuery to handle AJAX calls and DOM manipulation. It will break if JQuery is not included before it.

1.4.1 Project level settings

All the settings defined below will mostly be used when making AJAX calls and DOM manipulations. These settings will be supplied to include_notify_js_variables template tag for JS inclusion.

The settings will be represented in the following way

PROJECT_LEVEL_SETTING (default value) Description for the project level setting.

NOTIFY_SOFT_DELETE (True) Whether to do delete notifications softly or not.

NOTIFY_NF_LIST_CLASS_SELECTOR (.notifications) Class selector of element containing list of notification elements.

NOTIFY_SINGLE_NF_CLASS_SELECTOR (notifications) Class selector of individual notification elements.

NOTIFY_NF_BOX_CLASS_SELECTOR (.notification-box-list) Class selector of element containing list of notification elements on *notification box*.

NOTIFY_SINGLE_NF_BOX_CLASS_SELECTOR (.notification-box) Class selector of individual *notification-box* elements.

NOTIFY_MARK_NF_CLASS_SELECTOR (.mark-notification) Class selector for sub-element of notification element performing *mark* as read/unread action. This will be same for both, full page notifications as well as notification box on the navbar.

NOTIFY_MARK_ALL_NF_CLASS_SELECTOR (.mark-all-notifications) Class selector for sub-element of notification element performing *mark_all* as read/unread action.

NOTIFY_READ_NOTIFICATION_CSS (read) Class of parent notification element when its status is read.

NOTIFY_UNREAD_NOTIFICATION_CSS (unread) Class of parent notification element when its status is unread.

NOTIFY_DELETE_NF_CLASS_SELECTOR (.delete-notification) Class selector for sub-element of notification element performing *delete* action.

NOTIFY_UPDATE_TIME_INTERVAL (5000) Time interval (in ms) between ajax calls for notification update.

1.5 Understanding Notification Attributes

This is a detailed walk through the functioning of `django-notify-x`. This page will describe the use of Notification Model fields and the `notify` signal, so that you can make the most out of it.

1.5.1 Important fields in Notification Model

Anonymous values:

- Each activity stream component can carry an anonymous value instead of being a model object.
- You can also specify the its URL as text. By default `get_absolute_url()` method is called to assign a URL to activitiy stream participator.
- **At Django Model level inserts, the field name of these components are as follows:**
  - `actor_content_object`: For directly populating the model using a model instance.
  - `actor_content_type`: For manually assinging the object's content_type.
  - `actor_object_id`: ID of the object for manual assignment.
  - `actor_text`: Name of the actor in plain text (useful when using anonymous actor).

  - `actor_url_text`: URL of the actor in plain text (useful when using anonymous actor).
  - ... the same goes for **target and obj** components of activity stream.

Notification types:

- Each notification is different, they must be formatted differently during HTML rendering. For this, each notification gets to carry its own notification type.
- This notification type will be used to search the special template for the notification located at `notifications/includes/NF_TYPE.html` of your template directory.
- The main reason to add this field is to save you from the pain of writing `if...elif...else` blocks in your template file just for handling how notifications will get rendered.
- With this, you can just save template for an individual notification type and call the *template-tag* to render all notifications for you without writing a single `if...elif...else` block.
- You'll just need to do a `{% render_notifications using NOTIFICATION_OBJ %}` and you'll get your notifications rendered.
- By default, every `nf_type` is set to `default`.

The shortcut properties for activity components:

- Every activity component has its property which returns either the *__str__* value of `ACTIVITY-COMPONENT_content_object` or the value of its anonymous text or `None`.
  - For example:

```
notification = Notification.objects.get(pk=1)

# Instead of doing this
if notification.actor_content_object:
    actor_value = notification.actor_content_object
elif notification.actor_text:
    actor_value = notification.actor_text
else:
    actor_value = 'fallback text'

# you can do this
notification.actor
```

- The same way, every activity component is supposed contain a URL which is either the model object's `get_absolute_url()` value or `ACTIVITYCOMPONENT_url_text` or `None` or a fallback value.
  - You guessed it right!, There's a property for this thing too.
  - You can just access the activity component's URL like `notification.actor_url`.
  - `notification.actor_url` will either return the URL using the above methods or just return a `"#"` as a fallback URL.
- Not a property, but a model method...

  `.as_json()`, converts the values of the model instance to JSON or python dictonary, it is used when sending live notification updates as JSON format.

Other fields:

verb

Carries the verb of the notification performed.

description

Carries *optional* text description of the notification.

extra

JsonField, allows arbirary values in JSON format, so that you can store other useful information about a specific notification.

deleted

Useful when you want to *soft delete* notifications instead of diretly deleting them from database. The nature of this attribute can be controlled by a setting `NOTIFY_SOFT_DELETE = False`. This will delete notifications directly from database. By default, notifications are soft-deleted.

1.5.2 Important options in notify method

The keyword arguments when sending a `notify` signal are quite different than that of the Notification Model. Some fields are skipped and some are renamed for convenience. The below is the overview of the significant changes you need to know.

The actor, target and obj components

The `actor` keyword argument acts as `actor_content_object` at Django model level insert. There is not explicit *kwarg* for `actor_content_type` and `actor_object_id`. Same goes for `target` and `obj`.

The recipient and recipient_list

The `notify` signal takes these two as conditionally optional keyword arguments. They can neither be supplied together nor be empty. The `recipient` takes a user model instance only, the same case is with the `recipient_list` which takes a `list()` or `set()` of user model instances. A `set()` is preferred because it will only contain unique elements and eliminate duplicate notification checks and membership checks on a set are faster.

They're accessible from single signal because it would be highly redundant to create a separate signal with almost identical parameters just for the sake of making things distinguishable.

The first positional argument

When sending a notification, the first arument stands for the `sender` of the signal. For most cases it will be your `User` model. You can either user a user instance or the model class itself as the first parameter.

1.6 Notification and templates

Django's templating system plays a very important role in formatting your notifications without hassles. The templates for `django-notify-x` app are supposed to be stored in `notifications` directory of your default template directory.

As explained in previous chapter, `django-notify-x` uses templates to format different kinds of notifications corresponding to the `nf_type` of the notification.

1.6.1 Templates for notification types

If you want to render notifications using the `render_notifications` template tag and want to distingush notifications by their output locations (for example, *a typical web app has a notification page as well as a notification-box in it's navigation bar*.) you can use templates to store HTML format data for different kind of notifications.

The templates for notifications are stored in `notifications` directory of templates folder.

The tree structure representation of template directory

```
templates
|-- base.html
`-- notifications
    |-- all.html
    |-- base.html
    `-- includes
        |-- default_box.html
        |-- default.html
        |-- delete_success.js
        |-- js_variables.html
        |-- mark_all_success.js
        |-- mark_success.js
        |-- navbar.html
        `-- update_success.js
```

The `includes` sub-directory is for storing HTML output template for individual notifications and also Javascript files corresponding to the functions in the `notifyX.js` file.

For rendering notifications

The `all.html` and `base.html` under `notifications` directory are used when rendering the *all notifications page*. You can override both of them by creating them on your default template directory under `notifications` sub-directory.

For formatting notification

HTML output format for individual notifications are supposed to be stored in `notifications/includes/` with the name of their `nf_type` ending with a `.html`.

Note: If you're going to use live ajax notifications and you have a notification box to display instant notifications on your project, you might want to create one more file which starts with the corressponding `nf_type` and ends with `_box.html` in order to use different formatting.

For example, if you have notifications with `nf_type` set to `followed_user`, the name of the custom template files will be:

```
followed_user.html
followed_user_box.html
```

Put those files in `notifications/includes/` directory of your template directory.

For full page notifications, we'll try to find a template in the following order:

```
followed_user.html
default.html
```

For live ajax notifications, we'll try to find a template in the following order:

```
followed_user_box.html
default_box.html
followed_user.html
default.html
```

Contents of 'notifications/includes/followed_user.html':

```
<!-- this format is not compulsory, you can have HTML that suits your project -->
<li data-nf-id="{{ notification.id }}"
class="notification list-group-item {{ notification.read|yesno:'read,unread' }}">
    <a href="{{ notification.actor_url }}">{{ notification.actor }}</a> {{␣
↪notification.verb }}
    <span class="timesince">{{ notification.created|timesince }} ago</span>

    <button data-id="{{ notification.id }}" class="mark-notification"
        data-mark-action="{{ notification.read|yesno:'unread,read' }}"
        data-toggle-text="Mark as {{ notification.read|yesno:_('read,unread') }}">

        Mark as {{ notification.read|yesno:'unread,read' }}

    </button>
    <button class="delete-notification" data-id="{{ notification.id }}">X</button>
</li>
```

Contents of 'notifications/includes/followed_user_box.html':

```
<!-- this format is not compulsory, you can have HTML that suits your project -->
<li data-nf-id="{{ notification.id }}"
class="notification list-group-item {{ notification.read|yesno:'read,unread' }}">
    <a href="{{ notification.actor_url }}">{{ notification.actor }}</a> {{␣
↪notification.verb }}
    <span class="timesince">{{ notification.created|timesince }} ago</span>

    <button data-id="{{ notification.id }}" class="mark-notification"
        data-mark-action="{{ notification.read|yesno:'unread,read' }}"
        data-toggle-text="Mark as {{ notification.read|yesno:_('read,unread') }}">

        Mark as {{ notification.read|yesno:'unread,read' }}

    </button>
    <button class="delete-notification" data-id="{{ notification.id }}">X</button>
</li>
```

Note: The contents of the examples above are identical, in this case you might create only the *followed_user.html* file.

Things to take care when writing notification templates

Other than what we just discussed above, we need to make sure we do the following things correctly inorder to make this app work. These things are mostly the html attribute values will will be used by the javascript file inorder to perform DOM manipulation/

data-nf-id Attribute assigned to the parent element of notification. This will help our javascript to correctly select the parent notification element.

data-mark-action & data-id Attribute assigned to an element which will handle the control for marking a notification as read or unread. `data-mark-action` will also be used when marking all notifications as read or unread.

data-mark-action & data-toggle-text The element that holds the attribute `data-mark-action` will have it's innerHTML text replaced to reflect the toggle behavior. In order to customize the toggled text, you should provide the opposite text into a `data-toggle-text` attribute.

delete-notification & data-id Attribute assigned to an element which handles deleting of a notification.

Note: The above settings are only necessary if you want things happen over AJAX. If you want to control things with POST request, there is absolutely no need of keeping these attributes.

Switching to bootstrap github-style notifications

As long as bootstrap v3 stylesheets and javascript are included in your project you can switch to using a github-style notification system using the included `bootstrap-style` folder. Simply `cp -R /path/to/notify/templates/bootstrap-style/ /path/to/project/templates/notifications` and it will override the default templates. Include the following javascript snippet on the page to activate the bootstrap nav tabs.

snippet:: {% load staticfiles %} <script src="{% static 'notify/notifyX.min.js' %}"></script> <script type="text/javascript">

$('#notification-tabs a').click(function (e) { e.preventDefault() $(this).tab('show')

})

</script>

1.6.2 Notification Template tags

This app comes with two notification tags, one renders notifications for you and the other includes javascript variables and functions relating the `notifyX.js` file.

render_notifications

As its name reflects, it will render notifications for you. `render_notifications` will take at least one parameter and maximum two parameters.

You can use them to render notifications using a `Notification` QuerySet object, like this:

```
{% load notification_tags %}
{% render_notifications using request.user.notifications.active %}
```

By default, the above tag will render notifications on the notifications page and not on the notification box. So it will use a template corresponing to it's `nf_type` with a `.html` suffix nothing more.

To render notificatons on a notifications box:

```
{% load notification_tags %}
{% render_notifications using request.user.notifications.active for box %}
```

This tag will look for template name with `_box.html` suffixed when rendering notification contents.

The `request.user.notifications.active` is just used to show an example of notification queryset, you can use any other way to supply a QuerySet of your choice.

As each notification has many generic relations (actor, target, object), and the Django's default behavior when evaluating queries is to hit the database once per relation per record, the amount of queries to render many notifications can grow quickly. To handle this case, the `Notification` queryset has a method `prefetch`, that prefetches the relations and reduces the number of queries needed to `N+Y`, where `N` is the number of notifications on the master queryset, and `Y` is the number of distinct models that your notifications refers to.

Use `prefetch` to reduce the number of queries:

```
{% load notification_tags %}
{% render_notifications using request.user.notifications.active.prefetch %}
```

include_notify_js_variables

This tag uses `notifications/includes/js_variables.html` to include a template populated with JS variables and functions. You can override the values of any JS variables by creating your own version of `js_variables.html` template.

To include JS variables for AJAX notification support, do this:

```
{% load notification_tags %}
{% include_notify_js_variables %}
```

This template inclusion includes four javascript files from the template includes directory, they are:

```
mark_success.js
mark_all_success.js
delete_success.js
update_success.js
```

All of them are nothing but javascript function declarations which are supposed to run when a JQuery AJAX request is successfully completed.

Note: In the previous versions, it was necessarty to add notification check before including the JS variables using the `include_notify_js_variables` template tag. It is no more required. The new update checks for authenticated users and then renders the tempalte if required.

---

user_notifications

Note: The `user_notifications` tag is a shortcut to the `render_notifications` tag. It directly renders the notifications of the logged-in user on the specified target.

You can use this tag like this:

```
{% load notification_tags %}
{% user_notifications %}
```

This tag renders active notifications of the user by using something like `request.user.notifications.active()`.

Just like `render_notifications` it also takes rendering target as an optional argument. You can specify rendering target like this:

```
{% load notification_tags %}
{% user_notifications for box %}
```

By default, it'll use 'page' as the rendering target and use full page notification rending template corresponding to the `nf_type` of the template.

1.7 The client-side flow

All the views of `django-notify-x` are compatible with both, AJAX and non-AJAX requests. The included javascript file helps you control the result of actions performed by user. Common tasks like making notifications fade out, changing background color when a notification is mark as read/ unread are already handled by default javascript functions.

Of course, you want things to appear differently for your project, for doing so, you can simply write your own version of the javascript functions and save them in the corresponding templates diretory of app and the things will get overridden easily.

To utilize read-only notifications or unread-only notifications, where the notification status is not toggled but set to read or unread respectively the list selector attribute class needs to be set to `.read-notifications` or `.unread-notifications`. By default the included template will set the `ul` element class to `.notifications` which results in the toggle of read/unread. You can create your own `all.html` template or use javascript to modify the class selector on the client side.

The contents of default javascript function-files may seem to be useless, may be they are. But the files are just to give you idea about how things are supposed to work.

See also:

You might want to have a look on Important HTML attributes of a notification templates. They'll play an important role in AJAX an DOM manipuation of notifications.

1.7.1 How notifications are updated

The notification_update view explains the total flow of notification update requests.

The AJAX requests for updation of notifications are fired in every 5 seconds by default. You can control the time interval between the update requests by adding an entry named `NOTIFY_UPDATE_TIME_INTERVAL` set you the number of milliseconds you want.

1.8 Models

class notify.models.**Notification**(**args*, ***kwargs*)

Notification Model for storing notifications. (Yeah, too obvious)

This model is pretty-much a replica of `django-notifications`'s model. The newly added fields just adds a feature to allow anonymous `actors`, `targets` and `object`.

Attributes:

recipient The user who receives notification.

verb Action performed by actor (not necessarily).

description Option description for your notification.

actor_text Anonymous actor who is not in content-type.

actor_url Since the actor is not in content-type, a custom URL for it.

. . . Same for target and obj.

nf_type

Each notification is different, they must be formatted differently during HTML rendering. For this, each notification gets to carry it own *notification type*.

This notification type will be used to search the special template for the notification located at `notifications/includes/NF_TYPE.html` of your template directory.

The main reason to add this field is to save you from the pain of writing `if...elif...else` blocks in your template file just for handling how notifications will get rendered.

With this, you can just save template for an individual notification type and call the *template-tag* to render all notifications for you without writing a single `if...elif...else block`.

You'll just need to do a `{% render_notifications using NOTIFICATION_OBJ %}` and you'll get your notifications rendered.

By default, every `nf_type` is set to `default`.

Extra **JSONField**, holds other optional data you want the notification to carry in JSON format.

Deleted Useful when you want to *soft delete* your notifications.

actor

Property to return actor object/text to keep things DRY.

Returns Actor object or Text or None.

actor_url
Property to return permalink of the actor. Uses `get_absolute_url()`.

If `get_absolute_url()` method fails, it tries to grab URL from `actor_url_text`, if it fails again, returns None (null).

Returns URL for the actor.

as_json()
Notification data in a Python dictionary to which later gets supplied to JSONResponse so that it gets JSON serialized the *django-way*

Returns Dictionary format of the QuerySet object.

static do_convert(*obj*)
Method to get string version of an object or set it to None conditionally. performs `force_text()` on the argument so that a foreignkey gets serialized? and spit out the `__str__` output instead of an Object.

Parameters **obj** – Object to convert.

Returns JSON-friendly data.

mark_as_read()
Marks notification as read

mark_as_unread()
Marks notification as unread.

obj
See `actor` property.

Returns Action Object or Text or None.

obj_url
See `actor_url` property.

Returns URL for Action Object.

target
See `actor` property

Returns Target object or Text or None

target_url
See `actor_url` property.

Returns URL for the target.

class notify.models.**NotificationQueryset**(*model=None*, *query=None*, *using=None*, *hints=None*)
Chain-able QuerySets using `.as_manager()`.

active()
QuerySet filter() for retrieving both read and unread notifications which are not soft-deleted.

Returns Non soft-deleted notifications.

active_all(*user=None*)
Method to soft-delete all notifications of a User (if supplied)

Parameters **user** – Notification recipient.

Returns Updates QuerySet as soft-deleted.

delete_all(*user=None*)
Method to soft-delete all notifications of a User (if supplied)

Parameters **user** – Notification recipient.

Returns Updates QuerySet as soft-deleted.

deleted()
QuerySet `filter()` for retrieving soft-deleted notifications.

Returns Soft deleted notification filter()

prefetch()
Marks the current queryset to prefetch all generic relations.

read()
QuerySet filter() for retrieving read notifications.

Returns Read and active Notifications filter().

read_all(*user=None*)
Marks all notifications as read for a user (if supplied)

Parameters **user** – Notification recipient.

Returns List of updated notifications with status codes

unread()
QuerySet filter() for retrieving unread notifications.

Returns Unread and active Notifications filter().

unread_all(*user=None*)
Marks all notifications as unread for a user (if supplied)

Parameters **user** – Notification recipient.

Returns List of updated notifications with status codes in form [{"status": 200, "id": "5"}, ..]

1.9 Views

notify.views.**delete**(*request*, **args*, ***kwargs*)
Deletes notification of supplied notification ID.

Depending on project settings, if `NOTIFICATIONS_SOFT_DELETE` is set to `False`, the notifications will be deleted from DB. If not, a soft delete will be performed.

By default, notifications are deleted softly.

Parameters **request** – HTTP request context.

Returns Response to delete action on supplied notification ID.

notify.views.**mark**(*request*, **args*, ***kwargs*)
Handles marking of individual notifications as read or unread. Takes `notification id` and mark `action` as POST data.

Parameters **request** – HTTP request context.

Returns Response to mark action of supplied notification ID.

notify.views.**mark_all**(*request*, **args*, ***kwargs*)
Marks notifications as either read or unread depending of POST parameters. Takes `action` as POST data, it can either be `read` or `unread`.

Parameters **request** – HTTP Request context.

Returns Response to mark_all action.

notify.views.**mark_read**(*request*, **args*, ***kwargs*)
Handles marking of individual notifications as read. Takes `notification id` as POST data.

Parameters **request** – HTTP request context.

Returns Response to mark action of supplied notification ID.

notify.views.**mark_unread**(*request*, **args*, ***kwargs*)
Handles marking of individual notifications as read. Takes `notification id` as POST data.

Parameters **request** – HTTP request context.

Returns Response to mark action of supplied notification ID.

notify.views.**notification_redirect**(*request*, *ctx*)
Helper to handle HTTP response after an action is performed on notification

Parameters

- **request** – HTTP request context of the notification
- **ctx** – context to be returned when a AJAX call is made.

Returns Either JSON for AJAX or redirects to the calculated next page.

notify.views.**notification_update**(*request*, **args*, ***kwargs*)
Handles live updating of notifications, follows ajax-polling approach.

Read more: http://stackoverflow.com/a/12855533/4726598

Required URL parameters: `flag`.

Explanation:

- The `flag` parameter carries the last notification ID received by the user's browser.
- This `flag` is most likely to be generated by using a simple JS/JQuery DOM. Just grab the first element of the notification list.
  - The element will have a `data-id` attribute set to the corresponding notification.
  - We'll use it's value as the flag parameter.
- The view treats the `last notification flag` as a model `filter()` and fetches all notifications greater than the flag for the user.
- Then the a JSON data is prepared with all necessary details such as, `verb`, `actor`, `target` and their URL etc. The foreignkey are serialized as their default `__str__` value.
  - Everything will be HTML escaped by django's `escape()`.
- Since these notification sent will only serve temporarily on the notification box and will be generated fresh using a whole template, to avoid client-side notification generation using the JSON data, the JSON data will also contain a rendered HTML string so that you can easily do a JQuery `$yourNotificationBox.prepend()` on the rendered html string of the notification.
- The template used is expected to be different than the template used in full page notification as the css and some other elements are highly likely to be different than the full page notification list.
- The template used will be the `notification type` of the notification suffixed `_box.html`. So, if your notification type is `comment_reply`, the template will be `comment_reply_box.html`.
  - This template will be stored in `notifications/includes/` of your template directory.

  - That makes: `notifications/includes/comment_reply_box.html`
- The rest is self-explanatory.

Parameters **request** – HTTP request context.

Returns Notification updates (if any) in JSON format.

`notify.views.`**`notifications`**(*request*, **args*, ***kwargs*)

Returns a rendered page of all notifications for the logged-in user. Uses `notification.nf_type` as the template for individual notification. Each notification type is expected to have a render template at `notifications/includes/NOTIFICATION_TYPE.html`.

Parameters **request** – HTTP request context.

Returns Rendered notification list page.

`notify.views.`**`read_and_redirect`**(*request*, **args*, ***kwargs*)

Marks the supplied notification as read and then redirects to the supplied URL from the `next` URL parameter.

IMPORTANT: This is CSRF - unsafe method. Only use it if its okay for you to mark notifications as read without a robust check.

Parameters

- **request** – HTTP request context.
- **notification_id** – ID of the notification to be marked a read.

Returns Redirect response to a valid target url.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index