django-tidings Documentation

Release latest

Erik Rose, Paul Craciunoiu, and the support.mozilla.com team

Feb 28, 2020

Contents

django-tidings is a framework for sending email notifications to users who have registered interest in certain events, such as the modification of some model object. Used by support.mozilla.org and developer.mozilla.org, it is optimized for large-scale installations. Its features include...

- Asynchronous operation using the celery task queue
- De-duplication of notifications
- Association of subscriptions with either registered Django users or anonymous email addresses
- Optional confirmation of anonymous subscriptions
- Hook points for customizing any page drawn and any email sent

CHAPTER 1

Contents

1.1 Installation

To install django-tidings in your Django project, make these changes to `settings.py`:

1. Add `tidings` to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
  'other',
  'apps',
  'here',
  ...
  'tidings'
]
```

2. Define the settings *TIDINGS_FROM_ADDRESS* and *TIDINGS_CONFIRM_ANONYMOUS_WATCHES*.

1.2 Introduction

Here we introduce django-tidings by way of examples and discuss some theory behind its design.

1.2.1 A Simple Example

On support.mozilla.com, we host a wiki which houses documents in 80 different human languages. For each document, we keep a record of revisions (in the standard wiki fashion) stretching back to the document's creation:

```
Document ---- Revision 1
          \__ Revision 2
          \__ Revision 3
          \__ ...
```

We let users register their interest in (or *watch*) a specific language, and they are notified when any document in that language is edited. In our "edit page" view, we explicitly let the system know that a noteworthy event has occurred, like so…

```
EditInLanguageEvent(revision).fire()
```

…which, if `revision`'s document was written in English, sends a mail to anyone who was watching English-language edits. The watching would have been effected through view code like this:

```
def watch_language(request):
    """Start notifying the current user of edits in the request's language."""
    EditInLanguageEvent.notify(request.user, language=request.locale)
    # ...and then render a page or something.
```

Thus we introduce the two core concepts of django-tidings:

Events Things that occur, like the editing of a document in a certain language

Watches Subscriptions. Specifically, mappings from events to the users or email addresses which are interested in them

Everything in tidings centers around these two types of objects.

1.2.2 Events, Watches, and Scoping

django-tidings is basically a big dispatch engine: something happens (that is, an *Event* subclass fires), and tidings then has to determine which *Watches* are relevant so it knows whom to mail. Each kind of event has an `event_type`, an arbitrary string that distinguishes it, and each watch references an event subclass by that string. However, there is more to the watch-event relationship than that; a watch has a number of other fields which can further refine its scope:

```
watch ---- event_type
       \__ content_type
       \__ object_id
       \__ 0..n key/value pairs ("filters")
```

In addition to an event type, a watch may also reference a content type, an object ID, and one or more *filters*, key/value pairs whose values come out of an enumerated set (no larger than integer space). The key concept in django-tidings, the one which gives it its flexibility, is that **only an Event subclass determines the meaning of its Watches' fields**. `event_type` always points to an Event subclass, but that is the only constant. `content_type` and `object_id` are almost always used as their names imply—but only by convention. And filters are designed from the start to be arbitrary.

As a user of django-tidings, you will be writing a lot of Event subclasses and deciding how to make use of Watch's fields for each. Let's take apart our simple example to see how the `EditInLanguageEvent` class might be designed:

```
1  class EditInLanguageEvent(Event):
2      """Event fired when any document in a certain language is edited
3
4      Takes a revision when constructed and filters according to that
5      revision's document's language
6
7      notify(), stop_notifying(), and is_notifying() take these args:
8
9          (user_or_email, language=some_language)
10
```

(continues on next page)

(continued from previous page)

```
    """
    event_type = 'edited wiki document in language'
    filters = set(['language'])  # for validation only

    def __init__(self, revision):
        super(EditInLanguageEvent, self).__init__()
        self.revision = revision

    def _users_watching(self, **kwargs):
        return self._users_watching_by_filter(
            language=self.revision.document.language,
            **kwargs)


    ...
```

This event makes use of only two *Watch* fields: the `event_type` (which is implicitly handled by the framework) and a filter with the key "language". `content_type` and `object_id` are unused. The action happens in the `_users_watching()` method, which *Event.fire()* calls to determine whom to mail. Line 20 calls *_users_watching_by_filter()*, which is the most interesting method in the entire framework. In essence, this line says "Find me all the watches matching my `event_type` and having a 'language' filter with the value `self.revision.document.language`." (It is always a good idea to pass `**kwargs` along so you can support the *exclude* option.)

Watch Filters

This is a good point to say a word about *WatchFilters*. A filter is a key/value pair. The key is a string and goes into the database verbatim. The value, however, is only a 4-byte unsigned int. If you pass a string as a watch filter value, it will be hashed to make it fit. Thus, watch filters are no good for *storing* data but only for distinguishing among members of enumerated sets.

An exception is if you pass an integer as a filter value. The framework will notice this and let the int through unmodified. Thus, you can put (unchecked) integer foreign key references into filters quite happily.

Details of the hashing behavior are documented in *hash_to_unsigned()*.

Wildcards

Think back to our *notify()* call:

```
EditInLanguageEvent.notify(request.user, language=request.locale)
```

It tells the framework to create a watch with the `event_type` `'edited wiki document in locale'` (tying it to `EditInLanguageEvent`) and a filter mapping "language" to some locale.

Now, what if we had made this call instead, omitting the `language` kwarg?

```
EditInLanguageEvent.notify(request.user)
```

This says "`request.user` is interested in *every* `EditInLanguageEvent`, regardless of language", simply by omission of the "language" filter. A similar logic applies to events which use the `content_type` or `object_id` fields: leave them blank in a call to *notify()*, and the user will watch events with any value of them.

If, for some odd reason, a user ends up watching both *all* `EditInLanguageEvents` and German `EditInLanguageEvents` in particular, never fear: he will not receive two mails every time someone edits a German article. tidings will automatically de-duplicate users within the scope of one event class. Also, when faced

with a registered user and an anonymous subscription having the same email address, tidings will favor the registered user. That way, any mails you generate will have the opportunity to use a nice username, etc.

1.2.3 Completing the Event Implementation

A few more methods are necessary to get to a fully working *EditInLanguageEvent*. Let's add them now:

```
class EditInLanguageEvent(Event):

    # Previous methods here

    def _mails(self, users_and_watches):
        """Construct the mails to send."""
        document = self.revision.document

        # This loop is shown for clarity, but in real code, you should use
        # the tidings.utils.emails_with_users_and_watches convenience
        # function.
        for user, watches in users_and_watches:
            yield EmailMessage(
                'Notification: an edit!',
                'Document %s was edited.' % document.title,
                settings.TIDINGS_FROM_ADDRESS,
                [user.email])

    @classmethod
    def _activation_email(cls, watch, email):
        """Return an EmailMessage to send to anonymous watchers.

        They are expected to follow the activation URL sent in the email to
        activate their watch, so you should include at least that.

        """
        return EmailMessage(
            'Confirm your subscription',
            'Click the link if you really want to subscribe: %s' % \
                cls._activation_url(watch)
            settings.TIDINGS_FROM_ADDRESS,
            [email])

    @classmethod
    def _activation_url(cls, watch):
        """Return a URL pointing to a view that activates the watch."""
        return reverse('myapp.activate_watch', args=[watch.id, watch.secret])
```

Default implementations of *`_activation_email()`* and *`_activation_url()`* are coming in a future version of tidings.

1.2.4 Watching an Instance

Often, we want to watch for changes to a specific object rather than a class of them. tidings comes with a purpose-built abstract superclass for this, *`InstanceEvent`*.

In the support.mozilla.com wiki, we allow a user to watch a specific document. For example…

```
EditDocumentEvent.notify(request.user, document)
```

With the help of *InstanceEvent*, this event can be implemented just by choosing an `event_type` and a `content_type` and, because we need Revision info in addition to Document info when we build the mails, overriding `__init__()`:

```
class EditDocumentEvent(InstanceEvent):
    """Event fired when a certain document is edited"""
    event_type = 'wiki edit document'
    content_type = Document

    def __init__(self, revision):
        """This is another common pattern: we need to pass the Document to
        InstanceEvent's constructor, but we also need to keep the new
        Revision around so we can pull info from it when building our
        mails."""
        super(EditDocumentEvent, self).__init__(revision.document)
        self.revision = revision

    def _mails(self, users_and_watches):
        # ...
```

For more detail, see the *InstanceEvent* documentation.

1.2.5 De-duplication

We have already established that *mails get de-duplicated within the scope of one event class*, but what about across many events? What happens when a document is edited and some user was watching both it specifically and its language in general? Does he receive two mails? Not if you use *EventUnion*.

When your code does something that could cause both events to happen, the naive approach would be to call them serially:

```
EditDocumentEvent(revision).fire()
EditInLanguageEvent(revision).fire()
```

That *would* send two mails. But if we use the magical *EventUnion* construct instead...

```
EventUnion(EditDocumentEvent(revision), EditInLanguageEvent(revision)).fire()
```

...tidings is informed that you're firing a bunch of events, and it sends only one mail.

A few notes:

- The *_mails()* method from the first event class passed is the one that's used, though you can change this by subclassing *EventUnion* and overriding its *_mails()*.
- Like the single-event de-duplication, *EventUnion* favors registered users over anonymous email addresses.

1.2.6 The Container Pattern

One common case for de-duplication is when watchable objects contain other watchable objects, as in a discussion forum where users can watch both threads and entire forums:

```
forum ---- thread
       \__ thread
       \__ thread
```

In this case, we might imagine having a `NewPostInThreadEvent` through which users watch a thread and a `NewPostInForumEvent` through which they watch a whole forum. Both events would be *`InstanceEvent`* subclasses:

```
1   class NewPostInForumEvent(InstanceEvent):
2       event_type = 'new post in forum'
3       content_type = Forum
4
5       def __init__(self, post):
6           super(NewPostInForumEvent, self).__init__(post.thread.forum)
7           # Need to store the post for _mails
8           self.post = post
9
10
11  class NewPostInThreadEvent(InstanceEvent):
12      event_type = 'new post in thread'
13      content_type = Thread
14
15      def __init__(self, post):
16          super(NewPostInThreadEvent, self).__init__(post.thread)
17          # Need to store the post for _mails
18          self.post = post
19
20      def fire(self, **kwargs):
21          """Notify not only watchers of this thread but of the parent forum as well."
    ↪""
22          return EventUnion(self, NewPostInForumEvent(self.post)).fire(**kwargs)
23
24      def _mails(self, users_and_watches):
25          return emails_with_users_and_watches(
26              'New post: %s' % self.post.title,
27              'forums/email/new_post.ltxt',
28              dict(post=post),
29              users_and_watches)
```

On line 20, we cleverly override `fire()`, replacing InstanceEvent's simple implementation with one that fires the union of both events. Thus, callers need only ever fire `NewPostInThreadEvent`, and it will take care of the rest.

Since `NewPostInForumEvent` will now be fired only from an *`EventUnion`* (and not as the first argument), it can get away without a `_mails` implementation. The container pattern is very slimming, both to callers and events.

1.2.7 Celery and Safe Asynchronous Tasks

Sending emails can be a slow process. By default, *`Event.fire()`* uses Celery to process the event asynchronously. The user's request is faster, and the emails can take as long as they need. This requires the pickle task serializer, which has security concerns. Celery 3.1 is the last version to enable pickle by default, and in Celery 4.0, JSON is the default serializer.

You can avoid using `pickle` by calling `fire()` synchronously:

```
MyEvent().fire(delay=False)
```

This will process the event and send any emails, which could take a long time. You can move event processing to the backend by writing your own task:

```
from celery.task import task

@task
def fire_myevent():
    MyEvent().fire(delay=False)


# Process an event
fire_myevent()
```

This can also be used with instance-based events, by loading the instance from the database inside of the task:

```
from celery.task import task
from myapp.models import Instance

@task
def fire_myinstanceevent(instance_id):
    instance = Instance.objects.get(instance_id)
    MyInstance(instance).fire(delay=False)


# Process an event
fire_myinstanceevent(instance.id)
```

This will allow you to process events asynchronously, and to use safer serializers like the JSON serializer.

1.3 Settings

django-tidings offers several Django settings to customize its behavior:

`django.conf.settings.`**`TIDINGS_FROM_ADDRESS`**

The address from which tidings' emails will appear to come. Most of the time, the *Event* has an opportunity to override this in code, but this setting is used as a default for conveniences like *emails_with_users_and_watches()* and the default implementation of *Event._activation_email()*.

Default: No default; you must set it manually.

Example:

```
TIDINGS_FROM_ADDRESS = 'notifications@example.com'
```

`django.conf.settings.`**`TIDINGS_CONFIRM_ANONYMOUS_WATCHES`**

A Boolean: whether to require email confirmation of anonymous watches. If this is `True`, tidings will send a mail to the creator of an anonymous watch with a confirmation link. That link should point to a view which calls *Watch.activate()* and saves the watch. (No such built-in view is yet provided.) Until the watch is activated, tidings will ignore it.

Default: No default; you must set it manually.

Example:

```
TIDINGS_CONFIRM_ANONYMOUS_WATCHES = True
```

django.conf.settings.**TIDINGS_MODEL_BASE**
A dotted path to a model base class to use instead of `django.db.models.Model`. This can come in handy if, for example, you would like to add memcached support to tidings' models. To avoid staleness, tidings will use the `uncached` manager (if it exists) on its models when performing a staleness-sensitive operation like determining whether a user has a certain watch.

Default: `'django.db.models.Model'`

Example:

```
TIDINGS_MODEL_BASE = 'sumo.models.ModelBase'
```

django.conf.settings.**TIDINGS_REVERSE**
A dotted path to an alternate implementation of Django's `reverse()` function. support.mozilla.com uses this to make tidings aware of the locale prefixes on its URLs, e.g. `/en-US/unsubscribe`.

Default: `'django.core.urlresolvers.reverse'`

Example:

```
TIDINGS_REVERSE = 'sumo.urlresolvers.reverse'
```

django.conf.settings.**TIDINGS_TEMPLATE_EXTENSION**
The extension for tidings view templates. It can be changed to support alternate template libraries like django-jinja. The extension is used in the *unsubscribe()* view:

- `'tidings/unsubscribe.'` + `TIDINGS_TEMPLATE_EXTENSION`
- `'tidings/unsubscribe_error.'` + `TIDINGS_TEMPLATE_EXTENSION`
- `'tidings/unsubscribe_success.'` + `TIDINGS_TEMPLATE_EXTENSION`

Default: `'html'`

Example:

```
TIDINGS_TEMPLATE_EXTENSION = 'jinja'
```

1.4 Views

If you wish to include unsubscribe links in your notification emails (recommended) and you happen to be using Jinja templates, you can point them to the provided *unsubscribe()* view:

tidings.views.**unsubscribe**(*request*, *watch_id*)
Unsubscribe from (i.e. delete) the watch of ID `watch_id`.

Expects an `s` querystring parameter matching the watch's secret.

GET will result in a confirmation page (or a failure page if the secret is wrong). POST will actually delete the watch (again, if the secret is correct).

Uses these templates:

- tidings/unsubscribe.html - Asks user to confirm deleting a watch
- tidings/unsubscribe_error.html - Shown when a watch is not found
- tidings/unsubscribe_success.html - Shown when a watch is deleted

The shipped templates assume a `head_title` and a `content` block in a `base.html` template.

The template extension can be changed from the default `html` using the setting *TIDINGS_TEMPLATE_EXTENSION*.

A stock anonymous-watch-confirmation view is planned for a future version of tidings.

1.5 Design Rationale

1.5.1 Explicit Event Firing

Events are manually fired rather than doing something implicit with, for example, signals. This is for two reasons:

1. In the case of events that track changes to model objects, we often want to tell the user exactly what changed. Pre- or post-save signals don't give us the original state of the object necessary to determine this, so we would have to backtrack, hit the database again, and just generally make a mess just to save one or two lines of event-firing code.
2. Implicitness could easily lead to accidental spam, such as during development or data migration.

If you still want implicitness, it's trivial to register a signal handler that fires an event.

1.6 Development Notes

1.6.1 Testing

To run django-tidings' tests, install tox and run it:

```
$ pip install tox
$ tox
```

1.6.2 Documentation

To build the docs, install Sphinx and run this:

```
make docs
```

1.7 Version History

Unreleased

- Drop support for Django 1.9, 1.10
- Add support for Django 2.1, 2.2, and 3.0
- Drop support for Python 3.4
- Add support for Python 3.7, 3.8

2.0.1 (2018-02-14)

- Fix a bug where asynchronously firing a task (the default) would raise an exception when run via Celery.

2.0 (2018-02-10)

- Added support for Django 1.9, 1.10, 1.11, and 2.0.
- Dropped support for Django 1.7 and South.
- Dropped support for jingo. Templates for the `unsubscribe` view are now standard Django templates.
- Added `Event.fire(delay=False)`, to avoid using the pickle serializer, which has security concerns.
- Added setting `TIDINGS_TEMPLATE_EXTENSION` to allow changing the template extension used by the `unsubscribe` view from `html` to `jinja`, `j2`, etc.
- Migrated Watch.email from a maximum length of 75 to 254, to follow the EmailField update in Django 1.8.

1.2 (2017-03-22)

- Added support for Django 1.8 and Python 3
- Dropped support for Python 2.6

1.1 (2015-04-23)

- Added support for Django 1.7
- Dropped support for Django 1.4, 1.5 and 1.6
- Dropped mock, Fabric and django-nose dependencies.
- Moved tests outside of app and simplified test setup.
- Added Travis CI: https://travis-ci.org/mozilla/django-tidings
- Moved to ReadTheDocs: https://django-tidings.readthedocs.io/en/latest/

1.0 (2015-03-03)

- Support Django 1.6.
- Fix a bug in reconstituting models under (perhaps) Django 1.5.x and up.
- Remove rate limit on `claim_watches` task.
- Add tox to support testing against multiple Django versions.

0.4

- Fix a deprecated celery import path.
- Add support for newer versions of Django, and drop support for older ones. We now support 1.4 and 1.5.
- Add an initial South migration.

Warning: If you're already using South in your project, you need to run the following command to create a "fake" migration step in South's migration history:

```
python path/to/manage.py migrate tidings --fake
```

0.3

- Support excluding multiple users when calling *`fire()`*.

0.2

- API change: *_mails()* now receives, in each user/watch tuple, a list of *Watch* objects rather than just a single one. This enables you to list all relevant watches in your emails or to make decisions from an *EventUnion*'s `_mails()` method based on what kind of events the user was subscribed to.
- Expose a few attribute docs to Sphinx.

0.1

- Initial release. In production on support.mozilla.com. API may change.

1.8 Reference Documentation

After understanding the basic concepts of tidings from the *Introduction*, these docstrings make a nice comprehensive reference.

1.8.1 events

class tidings.events.**Event**

Abstract base class for events

An *Event* represents, simply, something that occurs. A *Watch* is a record of someone's interest in a certain type of *Event*, distinguished by `Event.event_type`.

Fire an Event (`SomeEvent.fire()`) from the code that causes the interesting event to occur. Fire it any time the event *might* have occurred. The Event will determine whether conditions are right to actually send notifications; don't succumb to the temptation to do these tests outside the Event, because you'll end up repeating yourself if the event is ever fired from more than one place.

Event subclasses can optionally represent a more limited scope of interest by populating the `Watch.content_type` field and/or adding related *WatchFilter* rows holding name/value pairs, the meaning of which is up to each individual subclass. NULL values are considered wildcards.

Event subclass instances must be pickleable so they can be shuttled off to celery tasks.

classmethod _activation_email(*watch*, *email*)

Return an EmailMessage to send to anonymous watchers.

They are expected to follow the activation URL sent in the email to activate their watch, so you should include at least that.

classmethod _activation_url(*watch*)

Return a URL pointing to a view which *activates* a watch.

TODO: provide generic implementation of this before liberating. Generic implementation could involve a setting to the default `reverse()` path, e.g. `'tidings.activate_watch'`.

_mails(*users_and_watches*)

Return an iterable yielding an EmailMessage to send to each user.

Parameters **users_and_watches** – an iterable of (User or EmailUser, [Watches]) pairs where the first element is the user to send to and the second is a list of watches (usually just one) that indicated the user's interest in this event

emails_with_users_and_watches() can come in handy for generating mails from Django templates.

_users_watching(***kwargs*)

Return an iterable of Users and EmailUsers watching this event and the Watches that map them to it.

Each yielded item is a tuple: (User or EmailUser, [list of Watches]).

Default implementation returns users watching this object's event_type and, if defined, content_type.

_users_watching_by_filter(*object_id=None*, *exclude=None*, ***filters*)
Return an iterable of (User/*EmailUser*, [*Watch* objects]) tuples watching the event.

Of multiple Users/EmailUsers having the same email address, only one is returned. Users are favored over EmailUsers so we are sure to be able to, for example, include a link to a user profile in the mail.

The list of *Watch* objects includes both those tied to the given User (if there is a registered user) and to any anonymous Watch having the same email address. This allows you to include all relevant unsubscribe URLs in a mail, for example. It also lets you make decisions in the *_mails()* method of *EventUnion* based on the kinds of watches found.

"Watching the event" means having a Watch whose `event_type` is `self.event_type`, whose `content_type` is `self.content_type` or `NULL`, whose `object_id` is `object_id` or `NULL`, and whose WatchFilter rows match as follows: each name/value pair given in `filters` must be matched by a related WatchFilter, or there must be no related WatchFilter having that name. If you find yourself wanting the lack of a particularly named WatchFilter to scuttle the match, use a different event_type instead.

Parameters **exclude** – If a saved user is passed in as this argument, that user will never be returned, though anonymous watches having the same email address may. A sequence of users may also be passed in.

classmethod _validate_filters(*filters*)
Raise a TypeError if `filters` contains any keys inappropriate to this event class.

classmethod _watches_belonging_to_user(*user_or_email*, *object_id=None*, ***filters*)
Return a QuerySet of watches having the given user or email, having (only) the given filters, and having the event_type and content_type attrs of the class.

Matched Watches may be either confirmed and unconfirmed. They may include duplicates if the get-then-create race condition in *notify()* allowed them to be created.

If you pass an email, it will be matched against only the email addresses of anonymous watches. At the moment, the only integration point planned between anonymous and registered watches is the claiming of anonymous watches of the same email address on user registration confirmation.

If you pass the AnonymousUser, this will return an empty QuerySet.

classmethod description_of_watch(*watch*)
Return a description of the Watch which can be used in emails.

For example, "changes to English articles"

filters = {}
Possible filter keys, for validation only. For example: `set(['color', 'flavor'])`

fire(*exclude=None*, *delay=True*)
Notify everyone watching the event.

We are explicit about sending notifications; we don't just key off creation signals, because the receiver of a `post_save` signal has no idea what just changed, so it doesn't know which notifications to send. Also, we could easily send mail accidentally: for instance, during tests. If we want implicit event firing, we can always register a signal handler that calls *fire()*.

Parameters

- **exclude** – If a saved user is passed in, that user will not be notified, though anonymous notifications having the same email address may still be sent. A sequence of users may also be passed in.

- **delay** – If True (default), the event is handled asynchronously with Celery. This requires the pickle task serializer, which is no longer the default starting in Celery 4.0. If False, the event is processed immediately.

classmethod is_notifying(*user_or_email_*, *object_id=None*, ***filters*)
Return whether the user/email is watching this event (either active or inactive watches), conditional on meeting the criteria in `filters`.

Count only watches that match the given filters exactly–not ones which match merely a superset of them. This lets callers distinguish between watches which overlap in scope. Equivalently, this lets callers check whether *notify()* has been called with these arguments.

Implementations in subclasses may take different arguments–for example, to assume certain filters–though most will probably just use this. However, subclasses should clearly document what filters they supports and the meaning of each.

Passing this an `AnonymousUser` always returns `False`. This means you can always pass it `request.user` in a view and get a sensible response.

classmethod notify(*user_or_email_*, *object_id=None*, ***filters*)
Start notifying the given user or email address when this event occurs and meets the criteria given in `filters`.

Return the created (or the existing matching) Watch so you can call *activate()* on it if you're so inclined.

Implementations in subclasses may take different arguments; see the docstring of *is_notifying()*.

Send an activation email if an anonymous watch is created and *TIDINGS_CONFIRM_ANONYMOUS_WATCHES* is `True`. If the activation request fails, raise a ActivationRequestFailed exception.

Calling *notify()* twice for an anonymous user will send the email each time.

classmethod stop_notifying(*user_or_email_*, ***filters*)
Delete all watches matching the exact user/email and filters.

Delete both active and inactive watches. If duplicate watches exist due to the get-then-create race condition, delete them all.

Implementations in subclasses may take different arguments; see the docstring of *is_notifying()*.

class tidings.events.**EventUnion**(**events*)
Fireable conglomeration of multiple events

Use this when you want to send a single mail to each person watching any of several events. For example, this sends only 1 mail to a given user, even if he was being notified of all 3 events:

```
EventUnion(SomeEvent(), OtherEvent(), ThirdEvent()).fire()
```

__init__(**events*)

Parameters **events** – the events of which to take the union

_mails(*users_and_watches*)
Default implementation calls the *_mails()* of my first event but may pass it any of my events as `self`.

Use this default implementation when the content of each event's mail template is essentially the same, e.g. "This new post was made. Enjoy.". When the receipt of a second mail from the second event would add no value, this is a fine choice. If the second event's email would add value, you should probably fire both events independently and let both mails be delivered. Or, if you would like to send a single mail with a custom template for a batch of events, just subclass *EventUnion* and override this method.

_users_watching(***kwargs*)
Return an iterable of Users and EmailUsers watching this event and the Watches that map them to it.

Each yielded item is a tuple: (User or EmailUser, [list of Watches]).

Default implementation returns users watching this object's event_type and, if defined, content_type.

class tidings.events.**InstanceEvent**(*instance*, **args*, ***kwargs*)
Abstract superclass for watching a specific instance of a Model.

Subclasses must specify an `event_type` and should specify a `content_type`.

__init__(*instance*, **args*, ***kwargs*)
Initialize an InstanceEvent

Parameters **instance** – the instance someone would have to be watching in order to be notified when this event is fired.

_users_watching(***kwargs*)
Return users watching this instance.

classmethod is_notifying(*user_or_email*, *instance*)
Check if the watch created by notify exists.

classmethod notify(*user_or_email*, *instance*)
Create, save, and return a watch which fires when something happens to `instance`.

classmethod stop_notifying(*user_or_email*, *instance*)
Delete the watch created by notify.

exception tidings.events.**ActivationRequestFailed**(*msgs*)
Raised when activation request fails, e.g. if email could not be sent

1.8.2 models

class tidings.models.**EmailUser**(*email=''*)
An anonymous user identified only by email address.

This is based on Django's AnonymousUser, so you can use the `is_authenticated` property to tell that this is an anonymous user.

class tidings.models.**NotificationsMixin**(**args*, ***kwargs*)
Mixin for notifications models that adds watches as a generic relation.

So we get cascading deletes for free, yay!

class tidings.models.**Watch**(**args*, ***kwargs*)
The registration of a user's interest in a certain event

At minimum, specifies an event_type and thereby an *Event* subclass. May also specify a content type and/or object ID and, indirectly, any number of *WatchFilters*.

exception DoesNotExist

exception MultipleObjectsReturned

activate()
Enable this watch so it actually fires.

Return `self` to support method chaining.

content_type
Optional reference to a content type:

email
Email stored only in the case of anonymous users:

event_type
Key used by an Event to find watches it manages:

is_active
Active watches receive notifications, inactive watches don't.

secret
Secret for activating anonymous watch email addresses.

unsubscribe_url()
Return the absolute URL to visit to delete me.

class tidings.models.**WatchFilter**(**args*, ***kwargs*)
Additional key/value pairs that pare down the scope of a watch

exception DoesNotExist

exception MultipleObjectsReturned

value
Either an int or the hash of an item in a reasonably small set, which is indicated by the name field. See comments by *hash_to_unsigned()* for more on what is reasonably small.

tidings.models.**multi_raw**(*query*, *params*, *models*, *model_to_fields*)
Scoop multiple model instances out of the DB at once, given a query that returns all fields of each.

Return an iterable of sequences of model instances parallel to the `models` sequence of classes. For example:

```
[(<User such-and-such>, <Watch such-and-such>), ...]
```

1.8.3 tasks

tidings.tasks.**claim_watches**(*user*)
Attach any anonymous watches having a user's email to that user.

Call this from your user registration process if you like.

1.8.4 utils

tidings.utils.**hash_to_unsigned**(*data*)
If `data` is a string or unicode string, return an unsigned 4-byte int hash of it. If `data` is already an int that fits those parameters, return it verbatim.

If `data` is an int outside that range, behavior is undefined at the moment. We rely on the `PositiveIntegerField` on *WatchFilter* to scream if the int is too long for the field.

We use CRC32 to do the hashing. Though CRC32 is not a good general-purpose hash function, it has no collisions on a dictionary of 38,470 English words, which should be fine for the small sets that *WatchFilters* are designed to enumerate. As a bonus, it is fast and available as a built-in function in some DBs. If your set of filter values is very large or has different CRC32 distribution properties than English words, you might want to do your own hashing in your *Event* subclass and pass ints when specifying filter values.

tidings.utils.**emails_with_users_and_watches**(*subject*, *template_path*, *vars*, *users_and_watches*, *from_email='nobody@example.com'*, ***extra_kwargs*)

Return iterable of EmailMessages with user and watch values substituted.

A convenience function for generating emails by repeatedly rendering a Django template with the given `vars` plus a `user` and `watches` key for each pair in `users_and_watches`

Parameters

- **template_path** – path to template file
- **vars** – a map which becomes the Context passed in to the template
- **extra_kwargs** – additional kwargs to pass into EmailMessage constructor

1.8.5 views

tidings.views.**unsubscribe**(*request*, *watch_id*)

Unsubscribe from (i.e. delete) the watch of ID `watch_id`.

Expects an `s` querystring parameter matching the watch's secret.

GET will result in a confirmation page (or a failure page if the secret is wrong). POST will actually delete the watch (again, if the secret is correct).

Uses these templates:

- tidings/unsubscribe.html - Asks user to confirm deleting a watch
- tidings/unsubscribe_error.html - Shown when a watch is not found
- tidings/unsubscribe_success.html - Shown when a watch is deleted

The shipped templates assume a `head_title` and a `content` block in a `base.html` template.

The template extension can be changed from the default `html` using the setting *TIDINGS_TEMPLATE_EXTENSION*.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Credits

Erik Rose and Paul Craciunoiu developed django-tidings, replacing a simpler progenitor written by the whole support.mozilla.com team, including Ricky Rosario and James Socol.

Will Kahn-Greene worked on the 1.0 release. He updated the project to Django 1.4 through 1.6, added tox support to make multi-version testing easier, and fixed bugs.

Jannis Leidel worked on the 1.1 release. He updated the project to Django 1.7 and 1.8, added South migrations, refactored tests, added TravisCI and Coveralls support, switched from Fabric to a Makefile, switched from mock and django_nose to Django tests, and more.

John Whitlock worked on the 1.2 and 2.0 releases. He added support for Python 3 and Django 1.9 through 2.0. He added linting with flake8, switched from jingo to Django templates, and switched from django_celery to basic Celery.

Python Module Index

d

t

Index

Symbols

A

C

D

E

F

H

I

M

N

S

T

U

V

W