django-entity-event Documentation

Release 0.7.1

Erik Swanson

Aug 30, 2017

Contents

Django Entity Event is a framework for storing and rendering events, managing users subscriptions to those events, and providing clean ways to make notifying users as easy as possible. It builds on the Django Entity's powerful method of unifying individuals and groups into a consistent framework.

CHAPTER 1

Table of Contents

Installation

Django Entity Event is compatible with Python versions 2.7, 3.3, and 3.4.

Installation with Pip

Entity Event is available on PyPi. It can be installed using `pip`:

```
pip install django-entity-event
```

Use with Django

To use Entity Event with django, first be sure to install it and/or include it in your `requirements.txt` Then include `'entity_event'` in `settings.INSTALLED_APPS`. After it is included in your installed apps, run:

```
./manage.py migrate entity_event
```

if you are using South. Otherwise run:

```
./manage.py syncdb
```

Quickstart and Basic Usage

Django Entity Event is a great way to collect events that your users care about into a unified location. The parts of your code base that create these events are probably totally separate from the parts that display them, which are also separate from the parts that manage subscriptions to notifications. Django Entity Event makes separating these concerns as simple as possible, and provides convenient abstractions at each of these levels.

This quickstart guide handles the three parts of managing events and notifications.

1. Creating, and categorizing events.
2. Defining mediums and subscriptions.
3. Querying events and presenting them to users.

If you are not already using Django Entity, this event framework won't be particularly useful, and you should probably start by integrating Django Entity into your application.

Creating and Categorizing Events

Django Entity Event is structured such that all events come from a `Source`, and can be displayed to the user from a variety of mediums. When we're creating events, we don't need to worry much about what `Medium` the event will be displayed on, we do need to know what the `Source` of the events are.

`Source` objects are used to categorize events. Categorizing events allows different types of events to be consumed differently. So, before we can create an event, we need to create a `Source` object. It is a good idea to use sources to do fine grained categorization of events. To provide higher level groupings, all sources must reference a `SourceGroup` object. These objects are very simple to create. Here we will make a single source group and two different sources

```
from entity_event import Source, SourceGroup

yoursite_group = SourceGroup.objects.create(
    name='yoursite',
    display_name='Yoursite',
    description='Events on Yoursite'
)

photo_source = Source.objects.create(
    group=yoursite_group,
    name='photo-tag',
    display_name='Photo Tag',
    description='You have been tagged in a photo'
)

product_source = Source.objects.create(
    group=yoursite_group,
    name='new-product',
    display_name='New Product',
    description='There is a new product on YourSite'
)
```

As seen above, the information required for these sources is fairly minimal. It is worth noting that while we only defined a single `SourceGroup` object, it will often make sense to define more logical `SourceGroup` objects.

Once we have sources defined, we can begin creating events. To create an event we use the `Event.objects.create_event` method. To create an event for the "photo-tag" group, we just need to know the source of the event, what entities are involved, and some information about what happened

```
from entity_event import Event

# Assume we're within the photo tag processing code, and we'll
# have access to variables entities_tagged, photo_owner, and
# photo_location

Event.objects.create_event(
    source=photo_source,
    actors=entities_tagged,
    context={
        'photo_owner': photo_owner
        'photo_location': photo_location
    }
)
```

The code above is all that's required to store an event. While this is a fairly simple interface for creating events, in some applications it may be easier to read, and less intrusive in application code to use django-signals in the application code, and create events in signal handlers. In either case, We're ready to discuss subscription management.

Managing Mediums and Subscriptions to Events

Once the events are created, we need to define how the users of our application are going to interact with the events. There are a large number of possible ways to notify users of events. Email, newsfeeds, notification bars, are all examples. Django Entity Event doesn't handle the display logic for notifying users, but it does handle the subscription and event routing/querying logic that determines which events go where.

To start, we must define a `Medium` object for each method our users will consume events from. Storing `Medium` objects in the database has two purposes. First, it is referenced when subscriptions are created. Second the `Medium` objects provide an entry point to query for events and have all the subscription logic and filtering taken care of for you.

Like `Source` objects, `Medium` objects are simple to create

```
from entity_event import Medium

email_medium = Medium.objects.create(
    name="email",
    display_name="Email",
    description="Email Notifications"
)

newsfeed_medium = Medium.objects.create(
    name="newsfeed",
    display_name="NewsFeed",
    description="Your personal feed of events"
)
```

At first, none of the events we have been creating are accessible by either of these mediums. In order for the mediums to have access to the events, an appropriate `Subscription` object needs to be created. Creating a `Subscription` object encodes that an entity, or group of entities, wants to receive notifications of events from a given source, by a given medium. For example, we can create a subscription so that all the sub-entities of an `all_users` entity will receive notifications of new products in their newsfeed

```
from entity import EntityKind
from entity_event import Subscription

Subscription.objects.create(
    medium=newsfeed_medium,
    source=product_source,
    entity=all_users,
    sub_entity_kind=EntityKind.objects.get(name='user'),
    only_following=False
)
```

With this `Subscription` object defined, all events from the new product source will be available to the newsfeed medium.

If we wanted to create a subscription for users to get email notifications when they've been tagged in a photo, we will also create a `Subscription` object. However, unlike the new product events, not every event from the photos source is relevant to every user. We want to limit the events they receive emails about to the events where they are tagged in the photo.

In code above, you may notice the `only_following=False` argument. This argument controls whether all events are relevant for the subscription, or if the events are only relevant if they are related to the entities being subscribed. Since new products are relevant to all users, we set this to `False`. To create a subscription for users to receive emails about photos they're tagged in, we'll define the subscription as follows

```
Subscription.objects.create(
    medium=email_medium,
    source=photo_source,
    entity=all_users,
    sub_entity_kind=EntityKind.objects.get(name='user'),
    only_following=True
)
```

This will only notify users if an entity they're following is tagged in a photo. By default, entities follow themselves and their super entities.

Creating subscriptions for a whole group of people with a single entry into the database is very powerful. However, some users may wish to opt out of certain types of notifications. To accommodate this, we can create an `Unsubscription` object. These are used to unsubscribe a single entity from receiving notifications of a given source on a given medium. For example if a user wants to opt out of new product notifications in their newsfeed, we can create an `Unsubscription` object for them

```
from entity_event import Unsubscription

# Assume we have an entity, unsubscriber who wants to unsubscribe
Unsubscription.objects.create(
    entity=unsubscriber,
    source=product_source,
    medium=newsfeed_medium
)
```

Once this object is stored in the database, this user will no longer receive this type of notification.

Once we have `Medium` objects set up for the methods of sending notifications, and we have our entities subscribed to sources of events on those mediums, we can use the `Medium` objects to query for events, which we can then display to our users.

Querying Events

Once we've got events being created, and subscriptions to them for a given medium, we'll want to display those events to our users. When there are a large variety of events coming into the system from many different sources, it would be very difficult to query the `Event` model directly while still respecting all the `Subscription` logic that we hope to maintain.

For this reason, Django Entity Event provides three methods to make querying for events' to display extremely simple. Since the `Medium` objects you've created should correspond directly to a means by which you want to display events to users, there are three methods of the `Medium` class to perform queries.

1. `Medium.events`

2. `Medium.entity_events`
3. `Medium.events_targets`

Each of these methods return somewhat different views into the events that are being stored in the system. In each case, though, you will call these methods from an instance of `Medium`, and the events returned will only be events for which there is a corresponding `Subscription` object.

The `Medium.events` method can be used to return all the events for that medium. This method is useful for mediums that want to display events without any particular regard for who performed the events. For example, we could have a medium that aggregated all of the events from the new products source. If we had a medium, `all_products_medium`, with the appropriate subscriptions set up, getting all the new product events is as simple as

```
all_products_medium.events()
```

The `Medium.entity_events` method can be used to get all the events for a given entity on that medium. It takes a single entity as an argument, and returns all the events for that entity on that medium. We could use this method to get events for an individual entity's newsfeed. If we have a large number of sources creating events, with subscriptions between those sources and the newsfeed, aggregating them into one QuerySet of events is as simple as

```
newsfeed_medium.entity_events(user_entity)
```

There are some mediums that notify users of events independent of a pageview's request/response cycle. For example, an email medium will want to process batches of events, and need information about who to send the events to. For this use case, the `Medium.events_targets` method can be used. Instead of providing a `EventQueryset`, it provides a list of tuples in the form `(event, targets)`, where `targets` is a list of the entities that should receive that notification. We could use this function to send emails about events as follows

```
from django.core.mail import send_mail

new_emails = email_medium.events_targets(seen=False, mark_seen=True)

for event, targets in new_emails:
    send_mail(
        subject = event.context["subject"]
        message = event.context["message"]
        recipient_list = [t.entity_meta["email"] for t in targets]
    )
```

As seen in the last example, these methods also support a number of arguments for filtering the events based on properties of the events themselves. All three methods support the following arguments:

- `start_time`: providing a datetime object to this parameter will filter the events to only those that occurred at or after this time.
- `end_time`: providing a datetime object to this parameter will filter the events to only those that occurred at or before this time.
- `seen`: passing `False` to this argument will filter the events to only those which have not been marked as having been seen.
- `include_expired`: defaults to `False`, passing `True` to this argument will include events that are expired. Events with expiration are discussed in `create_event()`.
- `actor`: providing an entity to this parameter will filter the events to only those that include the given entity as an actor.

Finally, all of these methods take an argument `mark_seen`. Passing `True` to this argument will mark the events as having been seen by that medium so they will not show up if `False` is passed to the `seen` filtering argument.

Using these three methods with any combination of the event filters should make virtually any event querying task simple.

Advanced Features

The *Quickstart and Basic Usage* guide covers the common use cases of Django Entity Event. In addition to the basic uses for creating, storing, and querying events, there are some more advanced uses supported for making Django Entity Event more efficient and flexible.

This guide will cover the following advanced use cases:

- Dynamically loading context using `context_loader`
- Customizing the behavior of `only_following` by sub-classing `Medium`.

Rendering Events

Django Entity Event comes complete with a rendering system for events. This is accomplished by the setup of two different models:

1. `RenderingStyle`: Defines a style of rendering.
2. `ContextRenderer`: Defines the templates used for rendering, which rendering style it is, which source or source group it renders, and hints for fetching model PKs that are in event contexts.

When these models are in place, `Medium` models can be configured to point to a `rendering_style` of their choice. Events that have sources or source groups that match those configured in associated `ContextRenderer` models can then be rendered using the `render` method on the medium.

The configuration and rendering is best explained using a complete example. First, let's imagine that we are storing events that have contexts with information about Django User models. These events have a source called `user_logged_in` and track every time a user logs in. An example context is as follows:

```
{
    'user': 1, # The PK of the Django User model
    'login_time': 'Jan 10, 2014', # The time the user logged in
}
```

Now let's say we have a Django template, `user_logged_in.html` that looks like the following:

```
User {{ user.username }} logged in at {{ login_time }}
```

In order to render the event with this template, we first set up a rendering style. This rendering style is pretty short and could probably be displayed in many places that want to display short messages (like a notification bar). So, we can make a `short` rendering style as followings:

```
short_rendering_style = RenderingStyle.objects.create(
    name='short',
    display_name='Short Rendering Style')
```

Now that we have our rendering style, we need to create a context renderer that has information about what templates, source, rendering style, and context hints to use when rendering the event. In our case, it would look like the following:

```
context_renderer = ContextRenderer.objects.create(
    render_style=RenderingStyle.objects.get(name='short'),
    name='short_login_renderer',
    html_template_path='my_template_dir/user_logged_in.html',
    source=Source.objects.get(name='user_logged_in'),
    context_hints={
        'user': {
            'app_name': 'auth',
            'model_name': 'User',
        }
    }
)
```

In the above, we set up the context renderer to use the short rendering style, pointed it to our html template that we created, and also pointed it to the source of the event. As you can see from the html template, we want to reach inside of the Django User object and display the `username` field. In order to retrieve this information, we have told our context renderer to treat the `user` key from the event context as a PK to a Django `User` model that resides in the `auth` app.

With this information, we can now render the event using whatever medium we have set up in Django Entity Event.

```
notification_medium = Medium.objects.get(name='notification')
events = notification_medium.events()

# Assume that two events were returned that have the following contexts
# e1.context = {
#    'user': 1, # Points to Jeff's user object
#    'login_time': 'January 1, 2015',
# }
# e1.context = {
#     'user': 2, # Points to Wes's user object
#     'login_time': 'February 28, 2015',
# }
#
# Pass the events into the medium's render method
rendered_events = notification_medium.render(events)

# The results are a dictionary keyed on each event. The keys point to a tuple
# of text and html renderings.
print(rendered_events[0][1])
'jeff logged in at January 1, 2015'
print(rendered_events[1][1])
'wes logged in at February 28, 2015'
```

With the notion of rendering styles, the notification medium and any medium that can display short messages can utilize the renderings of the events. Other rendering styles can still be made for more complex renderings such as emails with special styling. For more advanced options on how to perform prefetch and select_relateds in the fetched contexts, view `ContextRenderer`.

Advanced Template Rendering Options

Along with the basic rendering capabilities, Django Entity Event comes with several other options and configurations for making rendering more robust.

Doing Prefetch and Select Related on Contexts

If you need to fetch additional relationships related to the model objects in the context data, a `select_related` key with a list of arguments can be provided to the dictionary of the model object you are fetching. The same is true for `prefetch_related` arguments as well. For example:

```
context_hints = {
    'account': {
        'app_name': 'my_account_app',
        'model_name': 'Account',
        'select_related': ['user'],  # Select the user object in the account model
        'prefetch_related': ['user__groups'],  # Prefetch user groups related to the account model
    }
}
```

Note that other context loaders can provide additional arguments to `select_related` and `prefetch_related`. Additional arguments provided by other context loaders will simply be unioned together when loading contexts of all events at once.

Passing Additional Context to Templates

Sometimes mediums need to have subtle differences in the rendering of their contexts. For example, headers might need to be added above and below a message or images might need to be displayed. For cases such as this, mediums come with an `additional_context` variable. Anything in this variable will always be passed into the context when events are rendered for that particular medium.

Using a Default Rendering Style

It can be cumbersome to set up context renderers for every particular rendering style when it isn't necessary. For example, sometimes tailored emails need a special rendering style, however, many events can be rendered in an email just fine with a simpler rendering style. For these cases, a user can set a Django setting called `DEFAULT_ENTITY_EVENT_RENDERING_STYLE` that points to the name of the default rendering style to use. If this variable is set and an appropriate context loader cannot be fetched for an event during rendering, the default rendering style will be used instead for that event (if it has been configured).

Serialized Context Data

If your display mechanism needs access to the context data of the event this can be accomplished by calling: `Event.get_serialized_context` method on the `Event` model. This will return a serializer safe version of the context that is used to generate the event output. This is useful if you want to make a completely custom rendering on the display device or you need additional context information about the event that occurred.

Customizing Only-Following Behavior

In the quickstart, we discussed the use of "only following" subscriptions to ensure that users only see the events that they are interested in. In this discussion, we mentioned that by default, entities follow themselves, and their super entities. This following relationship is defined in two methods on the `Medium` model: `Medium.followers_of` and `Medium.followed_by`. These two methods are inverses of each other and are used by the code that fetches events to determine the semantics of "only following" subscriptions.

It is possible to customize the behavior of these types of subscriptions by concretely inheriting from `Medium`, and overriding these two functions. For example, we could define a type of medium that provides the opposite behavior, where entities follow themselves and their sub-entities.

```
from entity import Entity, EntityRelationship
from entity_event import Medium

class FollowSubEntitiesMedium(Medium):
    def followers_of(self, entities):
        if isinstance(entities, Entity):
            entities = Entity.objects.filter(id=entities.id)
        super_entities = EntityRelationship.objects.filter(
            sub_entity__in=entities).values_list('super_entity')
        followed_by = Entity.objects.filter(
            Q(id__in=entities) | Q(id__in=super_entities))
        return followed_by

    def followed_by(self, entities):
        if isinstance(entities, Entity):
            entities = Entity.objects.filter(id=entities.id)
        sub_entities = EntityRelationship.objects.filter(
            super_entity__in=entities).values_list('sub_entity')
        followers_of = Entity.objects.filter(
            Q(id__in=entities) | Q(id__in=sub_entities))
        return followers_of
```

With these methods overridden, the behavior of the methods `FollowsubEntitiesMedium.events`, `FollowsubEntitiesMedium.entity_events`, and `FollowsubEntitiesMedium.events_targets` should all behave as expected.

It is entirely possible to define more complex following relationships, potentially drawing on different source of information for what entities should follow what entities. The only important consideration is that the `followers_of` method must be the inverse of the `followed_by` method. That is, for any set of entities, it must hold that

```
followers_of(followed_by(entities)) == entities
```

and

```
followed_by(followers_of(entities)) == entities
```

Code documentation

Contributing

Contributions and issues are most welcome! All issues and pull requests are handled through github on the ambitioninc repository. Also, please check for any existing issues before filing a new one. If you have a great idea but it involves big changes, please file a ticket before making a pull request! We want to make sure you don't spend your time coding something that might not fit the scope of the project.

Running the tests

To get the source source code and run the unit tests, run:

```
$ git clone git://github.com/ambitioninc/django-entity-event.git
$ cd django-entity-event
$ virtualenv env
$ . env/bin/activate
$ python setup.py install
$ coverage run setup.py test
$ coverage report --fail-under=100
```

While 100% code coverage does not make a library bug-free, it significantly reduces the number of easily caught bugs! Please make sure coverage is at 100% before submitting a pull request!

Code Quality

For code quality, please run flake8:

```
$ pip install flake8
$ flake8 .
```

Code Styling

Please arrange imports with the following style

```
# Standard library imports
import os

# Third party package imports
from mock import patch
from django.conf import settings

# Local package imports
from entity_event.version import __version__
```

Please follow Google's python style guide wherever possible.

Building the docs

When in the project directory:

```
pip install -r requirements/docs.txt
python setup.py build_sphinx
open docs/_build/html/index.html
```

Release Checklist

Before a new release, please go through the following checklist:

- Bump version in entity_event/version.py
- Add a release note in docs/release_notes.rst
- Git tag the version
- Upload to pypi:

  ```
  pip install wheel
  python setup.py sdist bdist_wheel upload
  ```

Vulnerability Reporting

For any security issues, please do NOT file an issue or pull request on github! Please contact security@ambition.com with the GPG key provided on Ambition's website.

Release Notes

v0.7.1

- Increase the uuid length

v0.7.0

- Add creation time for mediums so events can be queried per medium for after medium creation

v0.6.0

- Add python 3.5 support, remove django 1.7 support

v0.5.0

- Added django 1.9 support

v0.4.4

- Added some optimizations during event fetching to select and prefetch some related objects

v0.4.3

- Added ability to get a serialized version of an events context data

v0.4.0

- Added 1.8 support and dropped 1.6 support for Django

v0.3.4

- Fixed django-entity migration dependency for Django 1.6

v0.3.3

- Added Django 1.7 compatibility and app config

v0.3.2

- Added an additional_context field in the Medium object that allows passing of additional context to event renderings.
- Added ability to define a default rendering style for all sources or source groups if a context renderer is not defined for a particular rendering style.

v0.3.1

- Fixes a bug where contexts can have any numeric type as a pk

v0.3.0

- Adds a template and context rendering system to entity event

v0.2

- This release provides the core features of django-entity-event - Event Creation - Subscription Management - Event Querying - Admin Panel - Documentation

v0.1

- This is the initial release of django-entity-event.