Django Activity Stream Documentation

Release

Justin Quick

August 30, 2017

Contents

Django Activity Stream is a way of creating activities generated by the actions on your site.

It is designed for generating and displaying streams of interesting actions and can handle following and unfollowing of different activity sources. For example, it could be used to emulate the Github dashboard in which a user sees changes to projects they are watching and the actions of users they are following.

Action events are categorized by four main components.

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

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

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

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

Contents:

CHAPTER 1

Installation

Get the code

Installation is easy using `pip` and the only requirement is a recent version of Django.

```
$ pip install django-activity-stream
```

or get it from source

```
$ pip install git+https://github.com/justquick/django-activity-stream.git#egg=actstream
```

Basic app configuration

Then to add the Django Activity Stream to your project add the app `actstream` to your `INSTALLED_APPS` and urlconf.

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

Warning: In Django versions older than 1.7, the app must be placed somewhere after all the apps that are going to be generating activities (eg `django.contrib.auth`). The safest thing to do is to have it as the last app in `INSTALLED_APPS`.

Add the activity urls to your urlconf

```
urlpatterns = patterns('',
    ...
    ('^activity/', include('actstream.urls')),
    ...
)
```

The activity urls are not required for basic usage but provide activity *Feeds* and handle following, unfollowing and querying of followers.

Migrations

As of Django 1.7 and later, the core ships with an integrated migrations framework based on South migrations.

Note: In django-activity-streams 0.6.0 and later the migrations have been re-initialized with the newer migrations framework and the south migrations have been deprecated.

If you still wish to use the south migrations there is a way. Install it as you would normally and then modify your settings to use the deprecated south migration modules in actstream.

```
SOUTH_MIGRATION_MODULES = {
    'actstream': 'actstream.south_migrations',
}
```

Add extra data to actions

If you want to use custom data on your actions, then make sure you have django-jsonfield installed

```
$ pip install django-jsonfield
```

You can learn more at *Adding Custom Data to your Actions*

Supported Environments

The following Python/Django versions and database configurations have been tested to work with the latest version of django-activity-stream.

- **PostgreSQL 9.1, 9.2 and 9.3**
  - **Psy** = psycopg2 2.6
  - **PCffi** = psycopg2cffi 2.6.1
- **MySQL 5.5 and 5.6**
  - **My** = MySQL-python 1.2.5
  - **PyMy** = PyMySQL 0.6.6
- **S** = Sqlite 3.7

	Py 2.6	Py 2.7	Py 3.2	Py 3.3	Py 3.4	PyPy 2	PyPy 3
Django 1.4	Psy/My/S	Psy/My/S				PCffi/My/S	
Django 1.5-1.8	Psy/My/S	Psy/My/S	Psy/PyMy/S	Psy/PyMy/S	Psy/PyMy/S	PCffi/My/S	My/S

CHAPTER 2

Configuration

Model Registration

In order to have your models be either an actor, target, or action object they must first be registered with actstream. In v0.5 and above, actstream has a registry of all actionable model classes. When you register them, actstream sets up certain GenericRelations that are required for generating activity streams.

You normally call register right after your model is defined (models.py) but you can call it anytime before you need to generate actions or activity streams.

```
# myapp/models.py
from actstream import registry

class MyModel(models.Model):
    ...

# Django < 1.7
registry.register(MyModel)
```

For Django versions 1.7 or later, you should use AppConfig.

```
# myapp/apps.py
from django.apps import AppConfig
from actstream import registry

class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        registry.register(self.get_model('MyModel'))

# myapp/__init__.py
default_app_config = 'myapp.apps.MyAppConfig'
```

Note: Introducing the registry change makes the `ACTSTREAM_SETTINGS['MODELS']` setting obsolete so please use the register functions instead.

Settings

Update these settings in your project's `settings.py`. As of v0.4.4, all settings are contained inside the `ACTSTREAM_SETTINGS` dictionary. Here is an example of what you can set in your `settings.py`

```
ACTSTREAM_SETTINGS = {
    'MANAGER': 'myapp.managers.MyActionManager',
    'FETCH_RELATIONS': True,
    'USE_PREFETCH': True,
    'USE_JSONFIELD': True,
    'GFK_FETCH_DEPTH': 1,
}
```

Note: In v0.5 and above, since only Django>=1.4 is supported all generic lookups fall back to QuerySet.prefetch_related so the `USE_PREFETCH` and `GFK_FETCH_DEPTH` settings have been deprecated.

Supported settings are defined below.

MANAGER

The action manager is the Django manager interface used for querying activity data from the database.

The Python import path of the manager to use for `Action.objects`. Add your own manager here to create custom streams. There can only be one manager class per Django project.

For more info, see *Writing Custom Streams*

Defaults to `actstream.managers.ActionManager`

FETCH_RELATIONS

Set this to `False` to disable `select_related` and `prefetch_related` when querying for any streams. When `True`, related generic foreign keys will be prefetched for stream generation (preferrable).

Defaults to `True`

USE_PREFETCH

Deprecated since version 0.5: This setting is no longer used (see note above).

Set this to `True` to forcefully enable `prefetch_related` (Django>=1.4 only). On earlier versions, the generic foreign key prefetch fallback contained within `actstream.gfk` will be enabled.

Defaults to whatever version you have.

USE_JSONFIELD

Set this setting to `True` to enable the `Action.data` JSONField for all actions. Lets you add custom data to any of your actions, see *Adding Custom Data to your Actions*

Defaults to `False`

GFK_FETCH_DEPTH

Deprecated since version 0.5: This setting is no longer used (see note above).

Number of levels of relations that `select_related` will perform. Only matters if you are not running `prefetch_related` (Django<=1.3).

Defaults to `0`

CHAPTER 3

Generating Actions

Generating actions can be done using Django signals. A special `action` signal is provided for creating the actions.

```
from django.db.models.signals import post_save
from actstream import action
from myapp.models import MyModel

# MyModel has been registered with actstream.registry.register

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

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

There are several ways to generate actions in your code. You can do it through custom forms or by overriding predefined model methods, such as Model.save(). More on this last option can be found here: <https://docs.djangoproject.com/en/dev/topics/db/models/#overriding-predefined-model-methods>.

The logic is to simply import the action signal and send it with your actor, verb, target, and any other important arguments.

```
from actstream import action
from myapp.models import Group, Comment

# User, Group & Comment have been registered with
# actstream.registry.register

action.send(request.user, verb='reached level 10')

...

group = Group.objects.get(name='MyGroup')
action.send(request.user, verb='joined', target=group)

...

comment = Comment.create(text=comment_text)
action.send(request.user, verb='created comment', action_object=comment, target=group)
```

Actions are stored in a single table in the database using Django's ContentType framework and GenericForeignKeys to create associations with different models in your project.

Actions are generated in a manner independent of how you wish to query them so they can be queried later to generate different streams based on all possible associations.

CHAPTER 4

Adding Custom Data to your Actions

As of v0.4.4, django-activity-stream now supports adding custom data to any Actions you generate. This uses a `data` JSONField on every Action where you can insert and delete values at will. This behavior is disabled by default but just set `ACTSTREAM_SETTINGS['USE_JSONFIELD'] = True` in your settings.py to enable it.

Note: This feature requires that you have django-jsonfield installed

You can send the custom data as extra keyword arguments to the `action` signal.

```
action.send(galahad, verb='answered', target=bridgekeeper,
    favorite_color='Blue. No, yel... AAAAAAA')
```

Now you can retrieve the data dictionary once you grab the action and manipulate it to your liking at anytime.

```
>>> action = Action.objects.get(verb='answered', actor=galahad, target=bridgekeeper)
>>> action.data['favorite_color']
... 'Blue. No, yel... AAAAAAA'
>>> action.data['quest'] = 'To seek the Holy Grail'
>>> action.save()
>>> action.data
... {'favorite_color': 'Blue. No, Green - AAAAAAA', 'quest': 'To seek the Holy Grail'}
```

Even in a template

```
You are {{ action.actor }} your quest is {{ action.data.quest }} and your favorite color is {{ actio
```

CHAPTER 5

Following/Unfollowing Objects

Creating or deleting the link between a `User` and any particular object is as easy as calling a function:

```
from actstream.actions import follow, unfollow

# Follow the group (where it is an actor).
follow(request.user, group)

# Stop following the group.
unfollow(request.user, group)
```

By default, `follow` only follows the object where it is an actor. To also include activity stream items where the object is the target or action_object, set the `actor_only` parameter to `False`:

```
# Follow the group wherever it appears in activity.
follow(request.user, group, actor_only=False)
```

You can also just make a request to the `actstream_follow` view while authenticated. The request can use either `GET` or `POST`.

```
curl -X GET http://localhost:8000/activity/follow/<content_type_id>/<object_id>/ # Follow
curl -X GET http://localhost:8000/activity/unfollow/<content_type_id>/<object_id>/?next=/blog/ # Unfo
```

If you wish to pass the `actor_only` parameter, the procedure is identical, only you will use `folow_all` and `unfolow_all` in your request. For example:

```
curl -X GET http://localhost:8000/activity/follow_all/<content_type_id>/<object_id>/ # Follow
curl -X GET http://localhost:8000/activity/unfollow_all/<content_type_id>/<object_id>/?next=/blog/ #
```

Then the current logged in user will follow the actor defined by `content_type_id` & `object_id`. Optional `next` parameter is URL to redirect to.

There is also a function `actstream.actions.unfollow` which removes the link and takes the same arguments as `actstream.actions.follow`

Now to retrive the follower/following relationships you can use the convient accessors

```
from actstream.models import following, followers

followers(request.user) # returns a list of Users who follow request.user
following(request.user) # returns a list of actors who request.user is following
```

To limit the actor models for the following relationship, just pass the model classes

```
from django.contrib.auth.models import User, Group

following(request.user, User) # returns a list of users who request.user is following
following(request.user, Group) # returns a list of groups who request.user is following
```

CHAPTER 6

Action Streams

Listings of actions are available for several points of view. All streams return a `QuerySet` of `Action` items sorted by `-timestamp`.

Using Builtin Streams

There are several builtin streams which cover the basics, but you are never limited to them. They are available as simple functions you can import from `actstream.models`. Some are also available on any instance of a registered model using a `GenericRelatedObjectManager` behind the scenes. The examples below show you all ways of accessing them.

User Streams

User streams are the most important, like your News Feed on github. Basically you follow anyone (or anything) on your site and their actions show up here. These streams take one argument which is a `User` instance which is the one doing the following (usually `request.user`).

If optional parameter `with_user_activity` is passed as `True`, the stream will include user's own activity like Twitter. Default is `False`

```
from actstream.models import user_stream

user_stream(request.user, with_user_activity=True)
```

Generates a stream of `Actions` from objects that `request.user` follows

Actor Streams

Actor streams show you what a particular actor object has done. Helpful for viewing "My Activities".

```
from actstream.models import actor_stream

actor_stream(request.user)
# OR
request.user.actor_actions.all()
```

Generates a stream of `Actions` where the `request.user` was the `actor`

Action Object Streams

Action object streams show you what actions a particular instance was used as the `action_objct`

```
from actstream.models import action_object_stream

action_object_stream(comment)
# OR
comment.action_object_actions.all()
```

Generates a stream of `Actions` where the `comment` was generated as the `action_object`

Target Streams

Action object streams show you what actions a particular instance was used as the `target`

```
from actstream.models import target_stream

target_stream(group)
# OR
group.target_actions.all()
```

Generates a stream of `Actions` where the `group` was generated as the `target`

Model Streams

Model streams offer a much broader scope showing ALL `Actions` from any particular model. Argument may be a class or instance of the model.

```
from actstream.models import model_stream

model_stream(request.user)
```

Generates a stream of `Actions` from all `User` instances.

Any Streams

Any streams shows you what actions a particular object was involved in either acting as the `actor`, `target` or `action_object`.

```
from actstream.models import any_stream

any_stream(request.user)
```

Generates a stream of `Actions` where `request.user` was involved in any part.

Writing Custom Streams

You can override and extend the Action manager `Action.objects` to add your own streams. The setting `ACTSTREAM_SETTINGS['MANAGER']` tells the app which manager to import and use. The builtin streams are defined in `actstream/managers.py` and you should check out how they are written. Streams must use the `@stream` decorator. They must take at least one argument which is a model instance to be used for reference when creating streams. Streams may return:

- `dict` - `Action` queryset parameters to be AND'd together
- `tuple` of `dicts` - tuple of `Action` queryset parameter dicts to be OR'd together
- `QuerySet` - raw queryset of `Action` objects

When returning a queryset, you do NOT need to call `fetch_generic_relations()` or `select_related(..)`.

Example

To start writing your custom stream module, create a file in your app called `myapp/managers.py`

```
# myapp/managers.py
from datetime import datetime

from django.contrib.contenttypes.models import ContentType

from actstream.managers import ActionManager, stream

class MyActionManager(ActionManager):

    @stream
    def mystream(self, obj, verb='posted', time=None):
        if time is None:
            time = datetime.now()
        return obj.actor_actions.filter(verb = verb, timestamp__lte = time)
```

If you havent done so already, configure this manager to be your default `Action` manager by setting the *MANAGER* setting.

This example defines a manager with one custom stream which filters for 'posted' actions by verb and timestamp.

Now that stream is available directly on the `Action` manager through `Action.objects.mystream` or from the `GenericRelation` on any actionable model instance.

```
from django.contrib.auth.models import User
from actstream.models import Action

user_instance = User.objects.all()[0]
Action.objects.mystream(user_instance, 'commented')
# OR
user_instance.actor_actions.mystream('commented')
```

CHAPTER 7

Templatetags

Start off your templates by adding the following load tag.

```
{% load activity_tags %}
```

Displaying Streams

You can use the `activity_stream` templatetag to render any of the builtin streams or your own custom streams. The first argument is the name of the stream method and then other arguments are passed from your templates into the stream functions.

```
{% activity_stream 'actor' user %}
{% for action in stream %}
    {% display_action action %}
{% endfor %}
```

You can also access custom streams by name in the same way. The tag puts the resulting queryset into a context variable which is by default simply called `stream` but you can customize the name by passing a value for the `as` keyword argument.

```
{% activity_stream 'mystream' request.user 'commented' as='mycomments' %}
{% for action in mycomments %}
    {% display_action action %}
{% endfor %}
```

Both examples above use the `display_action` templatetag which is an include tag which passes the `action` variable to `actstream/action.html`. You can override it to make it render as you would like.

Follow/Unfollow

There are two templatetags which are helpful for rendering information for following and unfollowing entities. The first is `follow_url` which returns the url you can hit to either follow the entity if you are not following it or unfollow if you are following it. There is also a `is_following` template filter which returns True if the user is following the given entity. The end result is a link that is a toggle.

```
<a href="{% follow_url other_user %}">
    {% if request.user|is_following:other_user %}
        stop following
    {% else %}
```

```
        follow
    {% endif %}
</a>
```

The code above will generate the url for the user to only follow actions where the object is the target. If you want the user to follow an object as both actor and target, you need to use the `follow_all_url` tag.

```
<a href="{% follow_all_url other_user %}">
    {% if request.user|is_following:other_user %}
        stop following
    {% else %}
        follow
    {% endif %}
</a>
```

CHAPTER 8

Feeds

The app supports feeds that support the Atom Activity Streams 1.0 and JSON Activity Streams 1.0 specifications.

Builtin Feed URLs

If you register the app with this URL prefix you can obtain the feeds using the URLs below.

```
url('^activity/', include('actstream.urls'))
```

User Streams

Shows user stream for currently logged in user.

```
/activity/feed/atom/
/activity/feed/json/
```

Any Streams

```
/activity/feed/<content_type_id>/<object_id>/atom/
/activity/feed/<content_type_id>/<object_id>/json/
```

Model Streams

```
/activity/feed/<content_type_id>/atom/
/activity/feed/<content_type_id>/json/
```

Custom JSON Feed URLs

Custom JSON feeds based on your custom streams registered by *Writing Custom Streams*

```
# myapp/urls.py
from actstream.feeds import CustomJSONActivityFeed
url(r'^feeds/mystream/(?P<verb>.+)/$',
    CustomJSONActivityFeed.as_view(name='mystream'))
```

Output

JSON

Here is some sample output of the JSON feeds. The formatting and attributes can be customized by subclassing `actstream.feeds.AbstractActivityStream`

```
{
    "totalItems": 1
    "items": [
        {
            "actor": {
                "id": "tag:example.com,2000-01-01:/activity/actors/13/2/",
                "displayName": "Two",
                "objectType": "my user",
                "url": "http://example.com/activity/actors/13/2/"
            },
            "target": {
                "id": "tag:example.com,2000-01-01:/activity/actors/2/1/",
                "displayName": "CoolGroup",
                "objectType": "group",
                "url": "http://example.com/activity/actors/2/1/"
            },
            "verb": "joined",
            "id": "tag:example.com,2000-01-01:/activity/detail/3/",
            "published": "2000-01-01T00:00:00Z",
            "url": "http://example.com/activity/detail/3/"
        }
    ]
}
```

ATOM

Here is some sample output of the ATOM feeds. They are based on the Django syndication framework and you can subclass `actstream.feeds.ActivityStreamsBaseFeed` or any of its subclasses to modify the formatting.

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns:activity="http://activitystrea.ms/spec/1.0/" xml:lang="en-us"
      xmlns="http://www.w3.org/2005/Atom">
    <title>Activity feed for your followed actors</title>
    <link href="http://example.com/actors/14/1/" rel="alternate"></link>
    <link href="http://example.com/feed/atom/" rel="self"></link>
    <id>http://example.com/actors/14/1/</id>
    <updated>2014-08-31T12:42:05Z</updated>
    <subtitle>Public activities of actors you follow</subtitle>
    <entry>
        <uri>http://example.com/detail/3/</uri>
        <link type="text/html" href="http://example.com/detail/3/"
              rel="alternate"></link>
        <activity:verb>joined</activity:verb>
        <published>2000-01-01T00:00:00Z</published>
        <id>tag:example.com,2000-01-01:/detail/3/</id>
        <title>Two joined CoolGroup 14 years, 8 months ago</title>
        <author>
            <id>tag:example.com,2000-01-01:/actors/14/2/</id>
            <activity:object-type>my user</activity:object-type>
```

```
            <name>Two</name>
        </author>
        <activity:target>
            <id>tag:example.com,2000-01-01:/actors/2/1/</id>
            <activity:object-type>group</activity:object-type>
            <title>CoolGroup</title>
        </activity:target>
    </entry>
</feed>
```

CHAPTER 9

Changelog

0.6.0

- Django 1.8 support
- Migrated to new migrations framework in Django core
- Improved db field indexing for models
- Optional django-generic-admin widgets integration (if installed)
- Minor templating and unicode fixes
- Admin displays public flag in list display
- Improved docs

0.5.1

- Coverage testing using coveralls.io
- Feeds refactoring to include JSON and custom feeds
- Added “any” builtin stream
- Following method bugfix
- Register method bugfix
- Is installed check bugfix
- Tests for nested app models
- Moar tests!
- Added actstream/base.html template for extensibility help

0.5.0

- Django 1.6 and 1.7 support
- Python 3 and PyPy support
- Added new activity_stream templatetag

- Dropping support for Django<=1.3 and rely on prefetch_related.
- Added register method for actionable models
- Dropped support for ACTSTREAM_SETTINGS['MODELS'] setting
- Added AppConf to support Django>=1.7

0.4.5

- Django 1.5 support including custom User model
- Translations and templates install fixes
- Fixes for MySQL migrations
- Tox testing for Py 2.6, 2.7 and Django 1.3, 1.4, 1.5
- Various other bug fixes

0.4.4

- Added support for custom Action data using JSONField.
- User of django.timezone.now when available.
- Templatetag fixes and removal of the follow_label tag.
- More tests
- Packaging fixes to include locale & migrations.
- App settings provided by ACTSTREAM_SETTINGS dictionary.
- Added following/followers to model accessors and views.

0.4.3

- Fixed default templatetags to not require auth.User ContentType
- Added actor_url templatetag

0.4.2

- Query improvement supporting Django 1.4 prefetch_related (falls back to it's own prefetch also for older Django versions)
- Admin fixes
- Packaging fixes
- Templatetag cleanup and documentation

0.4.1

- Templatetag updates
- Follow anything
- Test improvements
- Loads of fixes

0.4.0

- Scalability thanks to GFK lookup to prefetch actor, target & action_object in Action streams
- Limit number models that will be involved in actions
- Automagically adds GenericRelations to actionable models
- Generates Activity Stream 1.0 spec Atom feed
- Deletes orphaned actions when referenced object is deleted
- Custom, developer generated managers and streams
- I18N in unicode representation and through templating
- Sphinx Docs
- Duh, a changelog

CHAPTER 10

API

Action Manager

class actstream.managers.**ActionManager**

Default manager for Actions, accessed through Action.objects

action_object(*manager*, **args*, ***kwargs*)

Stream of most recent actions where obj is the action_object. Keyword arguments will be passed to Action.objects.filter

actor(*manager*, **args*, ***kwargs*)

Stream of most recent actions where obj is the actor. Keyword arguments will be passed to Action.objects.filter

any(*manager*, **args*, ***kwargs*)

Stream of most recent actions where obj is the actor OR target OR action_object.

model_actions(*manager*, **args*, ***kwargs*)

Stream of most recent actions by any particular model

public(**args*, ***kwargs*)

Only return public actions

target(*manager*, **args*, ***kwargs*)

Stream of most recent actions where obj is the target. Keyword arguments will be passed to Action.objects.filter

user(*manager*, **args*, ***kwargs*)

Stream of most recent actions by objects that the passed User obj is following.

Follow Manager

class actstream.managers.**FollowManager**

Manager for Follow model.

followers(*actor*)

Returns a list of User objects who are following the given actor (eg my followers).

following(*user*, **models*)

Returns a list of actors that the given user is following (eg who im following). Items in the list can be of any model unless a list of restricted models are passed. Eg following(user, User) will only return users following the given user

for_object(*instance*)
Filter to a specific instance.

is_following(*user*, *instance*)
Check if a user is following an instance.

Views

actstream.views.**respond**(*request*, *code*)
Responds to the request with the given response code. If `next` is in the form, it will redirect instead.

actstream.views.**follow_unfollow**(*request*, **args*, ***kwargs*)
Creates or deletes the follow relationship between `request.user` and the actor defined by `content_type_id`, `object_id`.

actstream.views.**stream**(*request*, **args*, ***kwargs*)
Index page for authenticated user's activity stream. (Eg: Your feed at github.com)

actstream.views.**followers**(*request*, *content_type_id*, *object_id*)
Creates a listing of `User``s that follow the actor defined by ``content_type_id`, `object_id`.

actstream.views.**following**(*request*, *user_id*)
Returns a list of actors that the user identified by `user_id` is following (eg who im following).

actstream.views.**user**(*request*, *username*)
`User` focused activity stream. (Eg: Profile page twitter.com/justquick)

actstream.views.**detail**(*request*, *action_id*)
`Action` detail view (pretty boring, mainly used for get_absolute_url)

actstream.views.**actor**(*request*, *content_type_id*, *object_id*)
`Actor` focused activity stream for actor defined by `content_type_id`, `object_id`.

actstream.views.**model**(*request*, *content_type_id*)
`Actor` focused activity stream for actor defined by `content_type_id`, `object_id`.

Feeds

class actstream.feeds.**AbstractActivityStream**
Abstract base class for all stream rendering. Supports hooks for fetching streams and formatting actions.

format(*action*)
Returns a formatted dictionary for the given action.

format_action_object(*action*)
Returns a formatted dictionary for the action object of the action.

format_actor(*action*)
Returns a formatted dictionary for the actor of the action.

format_item(*action*, *item_type='actor'*)
Returns a formatted dictionary for an individual item based on the action and item_type.

format_target(*action*)
Returns a formatted dictionary for the target of the action.

get_object(**args*, ***kwargs*)
Returns the object (eg user or actor) that the stream is for.

get_stream(**args*, ***kwargs*)
Returns a stream method to use.

get_uri(*action*, *obj=None*, *date=None*)
Returns an RFC3987 IRI ID for the given object, action and date.

get_url(*action*, *obj=None*, *domain=True*)
Returns an RFC3987 IRI for a HTML representation of the given object, action. If domain is true, the current site's domain will be added.

items(**args*, ***kwargs*)
Returns a queryset of Actions to use based on the stream method and object.

Atom

Compatible with Atom Activity Streams 1.0 spec

class `actstream.feeds.`**AtomUserActivityFeed**
Atom feed of Activity for a given user (where actions are those that the given user follows).

class `actstream.feeds.`**AtomModelActivityFeed**
Atom feed of Activity for a given model (where actions involve the given model as any of the entities).

class `actstream.feeds.`**AtomObjectActivityFeed**
Atom feed of Activity for a given object (where actions involve the given object as any of the entities).

JSON

Compatible with JSON Activity Streams 1.0 spec

class `actstream.feeds.`**AtomUserActivityFeed**
Atom feed of Activity for a given user (where actions are those that the given user follows).

class `actstream.feeds.`**AtomModelActivityFeed**
Atom feed of Activity for a given model (where actions involve the given model as any of the entities).

class `actstream.feeds.`**AtomObjectActivityFeed**
Atom feed of Activity for a given object (where actions involve the given object as any of the entities).

Actions

`actstream.actions.`**follow**(*user*, *obj*, *send_action=True*, *actor_only=True*, ***kwargs*)
Creates a relationship allowing the object's activities to appear in the user's stream.

Returns the created `Follow` instance.

If `send_action` is `True` (the default) then a `<user> started following <object>` action signal is sent. Extra keyword arguments are passed to the action.send call.

If `actor_only` is `True` (the default) then only actions where the object is the actor will appear in the user's activity stream. Set to `False` to also include actions where this object is the action_object or the target.

Example:

```
follow(request.user, group, actor_only=False)
```

actstream.actions.**unfollow**(*user*, *obj*, *send_action=False*)
Removes a "follow" relationship.

Set `send_action` to `True` (`False` is default) to also send a ``<user> stopped following <object>`` action signal.

Example:

```
unfollow(request.user, other_user)
```

actstream.actions.**is_following**(*user*, *obj*)
Checks if a "follow" relationship exists.

Returns True if exists, False otherwise.

Example:

```
is_following(request.user, group)
```

actstream.actions.**action_handler**(*verb*, ***kwargs*)
Handler function to create Action instance upon action signal call.

Decorators

actstream.decorators.**stream**(*func*)
Stream decorator to be applied to methods of an `ActionManager` subclass

Syntax:

```
from actstream.decorators import stream
from actstream.managers import ActionManager

class MyManager(ActionManager):
    @stream
    def foobar(self, ...):
        ...
```

Templatetags

Start off your templates by adding:

```
{% load activity_tags %}
```

actstream.templatetags.activity_tags.**activity_stream**(*context*, *stream_type*, **args*, ***kwargs*)
Renders an activity stream as a list into the template's context. Streams loaded by stream_type can be the default ones (eg user, actor, etc.) or a user defined stream. Extra args/kwargs are passed into the stream call.

```
{% activity_stream 'actor' user %}
{% for action in stream %}
    {% display_action action %}
{% endfor %}
```

actstream.templatetags.activity_tags.**is_following**(*user*, *actor*)

Returns true if the given user is following the actor

```
{% if request.user|is_following:another_user %}
    You are already following {{ another_user }}
{% endif %}
```

actstream.templatetags.activity_tags.**display_action**(*parser*, *token*)

Renders the template for the action description

```
{% display_action action %}
```

actstream.templatetags.activity_tags.**follow_url**(*parser*, *token*)

Renders the URL of the follow view for a particular actor instance

```
<a href="{% follow_url other_user %}">
    {% if request.user|is_following:other_user %}
        stop following
    {% else %}
        follow
    {% endif %}
</a>
```

actstream.templatetags.activity_tags.**follow_all_url**(*parser*, *token*)

Renders the URL to follow an object as both actor and target

```
<a href="{% follow_all_url other_user %}">
    {% if request.user|is_following:other_user %}
        stop following
    {% else %}
        follow
    {% endif %}
</a>
```

actstream.templatetags.activity_tags.**actor_url**(*parser*, *token*)

Renders the URL for a particular actor instance

```
<a href="{% actor_url request.user %}">View your actions</a>
<a href="{% actor_url another_user %}">{{ another_user }}'s actions</a>
```

CHAPTER 11

Indices and tables

- *genindex*
- *modindex*
- *search*

a

A

D

F

G

I

M

P

R

S

T

U