django-actrack Documentation

Release 1.1

Thomas Khyn

Jul 31, 2020

Contents

`django-actrack` is an activity tracker for the Django framework. It enables recording any activity by any actor, relative to any number of targets or related objects, with or without additional data. The activity can then be retrieved through feeds associated to any instance linked to any action(s).

It has been tested with Django 2.2.* and 3.0.* and their matching Python versions (3.5 to 3.8).

If you like django-actrack and find it useful, you may want to thank me and encourage future development by sending a few mBTC / mBCH / mBSV at this address: `1EwENyR8RV6tMc1hsLTkPURtn5wJgaBfG9`.

Documentation contents:

CHAPTER 1

Quick start

1.1 Installation

As straightforward as it can be, using pip:

```
pip install django-actrack
```

You then need to modify your `INSTALLED_APPS` settings:

- make sure it contains `django.contrib.contenttypes`
- add `'actrack'` and `gm2m`

1.2 First steps

All right, let's start tracking.

1.2.1 Logging activity

To track actions, the first things we need are ... actions. Let's generate and log some. We use the `actrack.log` function:

```
import actrack

actrack.log(user, 'had_lunch')
```

`user` can be a user model instance (for example an instance of `django.contrib.auth`'s `User` model) but it could as well be any instance of any model. It could be a train, for example (though trains usually don't have lunch).

You can also provide targets and related objects to add information to the action:

```
actrack.log(train, 'left_station', targets=[origin], related=[destination])
```

Or any relevant data as key-word arguments:

```
actrack.log(train, 'arrived', time=now())
```

OK, we've generated a few actions, let's see how we can retrieve them.

1.2.2 Tracking activity

`django-actrack` uses trackers to retrieve actions associated to instances. If you want the user `user` (here it needs to be an actual user, see below) to track all actions related to a given `train`, you can create a tracker using `actrack.track`:

```
actrack.track(user, train)
```

This creates a tracker entry in the database that will be used to retrieve every activity related to `train`. `train` could have been any other instance of any other model, or even a model class itself to follow any instance of that model, but `user` must be an instance of the `USER_MODEL` specified in the *Settings* (which defaults to `AUTH_USER_MODEL`_).

1.2.3 Retrieving activity

To retrieve every action matching this tracker, `django-actrack` can provide convenient accessors, provided you have connected the model to it beforehand using the `@actrack.connect` decorator:

```
@actrack.connect
class Train(models.Models):
    ...
```

'Connecting' `django-actrack` with a model will expose an `actions` attribute on every instance of the model:

```
# all the actions where the train is involved
all_train_actions = train.actions.all()

# actions where the train is involved as an actor, target or related object
train_actions_as_actor = train.actions.as_actor()
train_actions_as_target = train.actions.as_target()
train_actions_as_related = train.actions.as_related()
```

All the above will work for a given user instance or any instance which model has been connected to `django-actrack` via the `connect` decorator.

Additionally, for user instances, we can invoke:

```
user_feed = user.actions.feed()
```

And this will fetch all the actions related to all the objects the user is tracking (trains, airplanes, cars, anything ...)

Note: It is not always possible to use the `connect` decorator this way. The most common example is `django.contrib.auth.User`. We therefore use `connect` as a simple function, somewhere in our app (for example in an `AppConfig` subclass' `ready()` method) so that it is executed when Django starts:

```
actrack.connect(UserModel)
```

1.2.4 Next steps

Want to track more trains? Head to the *Advanced features* page to discover all the advanced stuff `django-actrack` can offer, or check out the *API* and the *Settings*.

CHAPTER 2

Advanced features

The *Quick start* section showed you how to log, track and retrieve activity related to given instances.

This section provides more details on `django-actrack` basic workflow and presents some of its more advanced features.

2.1 Action creation parameters

Check the API documentation for *actrack.log* to learn more about the additional parameters that it can accept.

2.2 Action handlers

For each action you are using in your code, you can create a subclass of `actrack.ActionHandler` with a corresponding `verb` class attribute that will be related to this action. An instance of this handler class will be attached to any `Action` object that is created or retrieved, as the `handler` attribute:

```
from actrack import ActionHandler

class MyActionHandler(ActionHandler):
   verb = 'my_action'

   def render(self, context):
      return 'I did that'

   def do_something(self):
      for t in self.action.targets.all():
         do_something_with_this_target(t)
```

Handlers are used to process the action. The only special methods are:

render Called when you call `render` on an Action instance. See *Rendering*

get_text Returns the text associated to the action

get_timeinfo Returns the time info of the action

get_context Returns a default rendering context for the action, should you need it for template rendering

combine(kwargs) [classmethod] See *Combination*

group(newer_kw, older_kw) [classmethod] See *Grouping*

See the actrack.handler module for default implementations.

You can of course override any of the above methods in the `ActionHandler` subclasses if you need to customise how certain actions should be rendered or combined.

2.3 Combination

Sometimes, actions should be combined. Either because 2 same actions with different arguments occurred at the same time, because two actions are redundant and should be merged, or for whatever app-dependant reason.

Only actions with the same actor and targets can be combined.

Action handlers can define custom `combine_with_[verb]` methods that determine what to do when a `verb` action is already in the queue. The method takes the keyword arguments that would be passed to the 'Action' constructor, and can make use of `self.queue`, a registry of all the previously added keyword arguments in this request. When this method returns `True`, the currently logged action is discarded. In this case, it is the responsibility of `combine_with_[verb]` to amend the action to which the discarded action is combined.

Note that the combination occurs when the action is logged. If an action is combined / discarded, it is not placed into the queue. The queue is saved to the database when a request finishes, after *Grouping* takes place.

2.4 Grouping

When the same action is repeated over a number of objects or on the same object, it is useless to show very similar actions a number of times.

`django-actrack` provides a way to check if an action that is being logged is similar to recent actions and, if it finds one, it amends it instead of creating a new one.

The definition of 'recent' can be changed by the `GROUPING_DELAY` setting, in seconds. Individually, it is also possible to change this delay or disable action grouping when calling `actrack.log` using the `grouping_delay` argument.

By default, an action is considered 'similar' if it has the same actor, and at least the same *targets* or *related* objects. This can be customized by overriding the `group` method in the `ActionHandler` subclass relative to the relevant action.

Grouping only occurs when the action queue is saved.

2.5 Deleted items

This is a great feature of `django-actrack`. If an object to which an action is related (the object can be the actor, a target or related object) is deleted, the action itself can either be deleted (if passing `use_del_items=False` to `actrack.connect`) or can remain. If it remains, its reference to the deleted item is replaced by a reference to an instance of a special model, that stores a verbose description of the deleted item.

For example, if the `train` instance is deleted (retired from the railway company's network, for example), the actions that had been generated beforehand refering to that `train` will not be deleted, and one will still be able to read when the train started and when it arrived.

To retrieve the verbose description, `django-actrack` first looks for a `deleted_item_description` method, calls it with no arguments and takes the returned string as the description. If that fails, it will simply evaluate the instance as a string using `str`.

The same thing exists for serialization. By default, the `serialization` field of the deleted item instance is populated with `{'pk': object.pk}` where `object` is the object being deleted. The value stored in `serialization` can be customized on a per-instance basis using the `deleted_item_serialization` method.

Warning: If you are logging an action involving an instance while deleting it (typically within a *pre_delete* or *post_delete* signal handler), you need to turn it into a 'deleted item' first. This can be done using the function *actrack.deletion.get_del_item* which takes the instance as an argument and returns a deleted item instance. Be careful, get_del_item creates an entry for a deleted item in the database, so make sure you call it only when you are actually deleting an instance

2.6 Read / unread actions

When the `TRACK_UNREAD` *setting* is set to `True`, `django-actrack` can make the distinction between read and unread actions.

When a new action is created, it is simply considered ad unread by all users.

An action's status can be retrieved using the `Action.is_unread_for` method, which takes a user as sole argument.

To update this status, you may use the `Action.mark_read_for(user, force)` method. `force` will override the `AUTO_READ` setting.

Alternatively, if `AUTO_READ` is `True`, an action can be marked as read when it is rendered, using its `render` method.

There are also classmethods on `Action` that implement the same functions on a sequence of actions: `bulk_is_unread_for`, `bulk_mark_read_for` and `bulk_render`. All of them take an ordered sequence of actions as first argument and return a list of booleans for the first two and strings for the third.

2.7 Rendering

Speaking about rendering, any action can be rendered through its `render` method. `Action.render` calls the action handler's `render` method, that can be overridden in subclasses of `ActionHandler`.

The `ActionHandler.get_context` method generates a useful default context dictionary from the attached action data.

CHAPTER 3

Settings

The settings must be stored in your Django project's `settings` module, as a dictionary name `ACTRACK`. This dictionary may contain the following items:

USER_MODEL The user model that should be used for the owners of the tracker instances. Defaults to Django's `AUTH_USER_MODEL`

ACTIONS_ATTR The name of the accessor for actions, that can be changed in case it clashes with one of your models' fields. Defaults to `'actions'`

TRACKERS_ATTR The name of the accessor for trackers, that can be changed in case it clashes with one of your models' fields. Defaults to `'trackers'`

DEFAULT_HANDLER The path to the default action handler class (used when a matching action handler is not found). Defaults to `'actrack.ActionHandler'`

TRACK_UNREAD Should unread actions be tracked? Defaults to `True`.

AUTO_READ Should actions be automatically marked as read when rendered? Defaults to `True`.

GROUPING_DELAY The time in seconds after which an action cannot be merged with a more recent one. When set to `-1`, grouping is disabled. When set to `0`, grouping occurs only on unsaved actions. Defaults to `0`

PK_MAXLENGTH The maximum length of the primary keys of the objects that will be linked to action (as targets or related). Defaults to `16`.

LEVELS A dictionary of logging levels. Defaults to:

```
{
    'NULL': 0,
    'DEBUG': 10,
    'HIDDEN': 20,
    'INFO': 30,
    'WARNING': 40,
    'ERROR': 50,
}
```

Note: The logging levels should have upper case names and their values must be small positive integers from 0 to 32767

The defined logging levels can, after initialization, be accessed under the `actrack.level` module. E.g. `actrack.level.INFO`.

DEFAULT_LEVEL The default level to use for logging. Defaults to `LEVELS['INFO']`

READABLE_LEVEL Below that logging level (strictly), an action cannot appear as unread and cannot be marked as read. Defaults to `LEVELS['INFO']`

CHAPTER 4

API

`django-actrack` exposes several functions, models and managers.

4.1 Functions

This section lists all the functions exposed by `django-actrack` and documents their keyword arguments.

4.1.1 actrack.log(actor, verb, **kwargs)

Mandatory arguments:

actor The instance that generates the activity. Can be any instance of any model, does not have to be a user.

verb A string identifying the action. Tip: make it meaningful. The verb is used to retrieve a matching *Action handlers* subclass

Optional keyword arguments:

targets A model instance or list of model instances being directly affected by the new action.

related A model instance or list of model instances being related to the new action.

Note: Technically, the `targets` and `related` object lists are redundant and they could be merged. However it can be meaningful or practical to split the objects in two groups, hence the distinction.

---

timestamp The timestamp that should be recorded for the action. If not provided, this default to now.

level The logging level of the new action. Logging levels can especially be used to filter actions that can be marked as unread. See the LEVELS, READABLE_LEVEL and DEFAULT_LEVEL *Settings*.

using The database to store the new action in.

grouping_delay If an action with the same verb has occurred within the last `grouping_delay` (in seconds), it is merged with the current one. If it is set to `0`, this prevents the action from being grouped. See *Grouping*. Defaults to `GROUPING_DELAY`.

other keywords any other keyword will be included in the action's data. They must only contain serializable data.

4.1.2 actrack.track(user, to_track, **kwargs)

`actrack.track` can be used either to create a tracker or modify an existing one. It can track model instances but also model classes.

user The user who should track actions concerning `to_track`. Must be an instance of the model defined by `AUTH_USER_MODEL`

to_track Actions relative to this model instance will appear in the `user`'s actions feed

log If set to `True`, the function will log an action with the verb 'started tracking'. Defaults to `False`

actor_only Will track actions only when the provided tracked object is the actor of an action. Default to `True`.

using The database to store the new tracker in.

verbs The verbs to track. Exclude any action that does not match the provide verbs. Defaults to any verb.

4.1.3 actrack.untrack(user, to_untrack, **kwargs)

Deletes a tracker object or deletes some verbs from its verbs set.

Mandatory arguments:

user See *actrack.track*

to_untrack The model instance to untrack

Optional keyword arguments:

log See *actrack.track*

verbs The verbs to stop tracking. If it is empty or equal to the current verbs set, no verb is to be tracked anymore and the tracker is deleted. Defaults to all verbs.

using See *actrack.track*

4.1.4 @actrack.connect or actrack.connect(model)

The `actrack.connect` decorator can be used with an optional argument:

use_del_items Should the model that is to be connected use the *deleted items* feature? Defaults to `True`.

4.2 Managers

4.2.1 The `actions` manager

We've seen in the *Quick start* that connecting a django Model using the `actrack.connect` decorator exposed an `actions` attribute on every instance of that Model. This `actions` attribute is actually a Django Manager that queries *Action* instances:

```
@actrack.connect
class MyModel(models.Models):
    ...

instance = MyModel()

# this returns a Manager to fetch actions
instance.actions
```

An `actions` manager has several useful methods:

`instance.actions.as_actor(\*\*kw)` All the actions where instance is the actor.

`instance.actions.as_target(\*\*kw)` All the actions where instance is among the targets.

`instance.actions.as_related(\*\*kw)` All the actions where instance is among the related objects.

`instance.actions.all()` Overrides the normal `all` method and returns all the actions where instance is either the actor or in the targets or related objects. It is a combination of the results of the 3 above methods.

`instance.actions.feed(\*\*kw)` The most useful accessor. This will work only if instance is a user, and will return all the instances that match all the trackers the user is associated with.

All these manager methods take keyword arguments to further filter the result queryset and only fetch the actions you want (verbs, timestamp . . .).

4.2.2 The `trackers` manager

In addition to the `actions` attribute, `actrack.connect` makes another helpful manager available: the `trackers`

```
# this returns a Manager to fetch Tracker instances
instance.trackers
```

`instance.tracker.tracking(\*\*kw)` All the trackers that are tracking the instance.

`instance.tracker.users(\*\*kw)` All the users who are tracking the instance (= the owners of the trackers tracking the instance returned by the above method).

`instance.tracker.owned(\*\*kw)` Works only if instance is a user, returns all the trackers owned by the instance.

`instance.tracker.tracked(\*models, \*\*kw)` Works only if instance is a user, returns all the objects (various types) tracked by the user. Be aware that if there are model class trackers, there can be model classes in the returned set.

`instance.tracker.all()` Overrides the normal `all` method. If instance is a user, will return a combination of `instance.tracker.owned()` and `instance.tracker.tracking`. If not, it returns the same as `instance.tracker.tracking`.

Similarly as above, these manager methods take keyword arguments to further filter the result queryset and only fetch the trackers you want (except `tracker.tracked` that returns instances of different models).

4.2.3 The default `Action` manager

Just a small word on the manager associated with the *Action* model: it has a special method that returns all the actions followed by a given tracker:

Action.objects.tracked_by(tracker, **kw) Fetches all the `Action` instances tracked by the tracker `tracker`.

4.3 Models

4.3.1 Action

The core model of `django-actrack`.

class `actrack.models.`**Action**(**args*, ***kwargs*)

An action initiated by an actor and described by a verb. An action may have: - target objects (affected by the action) - related objects (related to the action)

actor
The actor, can be anything

targets
The target objects, can contain several objects of different types

related
The related objects, can also contain several objects of different types

verb
The action's verb or identifier

level
The action's level

data
Data associated to the action (stored in a JSON field)

timestamp
The timestamp of the action, from which actions are ordered

is_unread_for(*user*)
Returns True if the action is unread for that user

mark_read_for(*user*, *force=False*)
Attempts to mark the action as read using the tracker's mark_read method. Returns True if the action was unread before To mark several actions as read, prefer the classmethod bulk_mark_read_for

render(*user=None*, *context=None*)
Renders the action, attempting to mark it as read if user is not None Returns a rendered string

classmethod bulk_is_unread_for(*user*, *actions*)
Does not bring any performance gains over Action.is_read method, exists for the sake of consistency with bulk_mark_read_for and bulk_render

classmethod bulk_mark_read_for(*user*, *actions*, *force=False*)
Marks an iterable of actions as read for the given user It is more efficient than calling the mark_read method on each action, especially if many actions belong to only a few followers

Returns a list `l` of booleans. If `actions[i]` was unread before the call to bulk_mark_read_for, `l[i]` is True

classmethod bulk_render(*actions=()*, *user=None*, *context=None*)
Renders an iterable actions, returning a list of rendered strings in the same order as `actions`

If `user` is provided, the class method will attempt to mark the actions as read for the user using Action.mark_read above

4.3.2 Trackers

`django-actrack` features two types of trackers. A `Tracker` model (which instances are stored in the database) and a non-persistent `TempTracker` class which is not actually a model but instead can be used to generate read-only queries on-the-fly.

class `actrack.models.`**Tracker**(**args*, ***kwargs*)

Action tracking object, so that a user can track the actions on specific objects

user

The user to which the tracker instance is attached

tracked

The tracked object

verbs

All the verbs that are tracked (when empty, that means 'all verbs')

actor_only

Should the tracker only track actions where the tracked object is the actor?

update_unread()

Retrieves the actions having occurred after the last time the tracker was updated and mark them as unread (bulk-add to unread_actions).

clean()

Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

clean_fields(*exclude=None*)

Clean all fields and raise a ValidationError containing a dict of all validation errors if any occur.

full_clean(*exclude=None*, *validate_unique=True*)

Call clean_fields(), clean(), and validate_unique() on the model. Raise a ValidationError for any errors that occur.

get_deferred_fields()

Return a set containing names of deferred fields for this instance.

matches(*action*)

Returns true if an action is to be tracked by the Tracker object

refresh_from_db(*using=None*, *fields=None*)

Reload field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)

Handle the parts of saving which should be done only once per save, yet need to be done in raw saves, too. This includes some sanity checks and signal sending.

The 'raw' argument is telling save_base not to save any parent models and not to do any changes to the values before save. This is used by fixture loading.

serializable_value(*field_name*)
Return the value of the field name for this instance. If the field is a foreign key, return the id value instead of the object. If there's no Field object with this name on the model, return the model attribute's value.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

validate_unique(*exclude=None*)
Check unique constraints on the model and raise ValidationError if any failed.

class actrack.models.**TempTracker**(*user*, *tracked*, *verbs=()*, *actor_only=True*, *last_updated=None*)
A tracker that is designed to be used 'on the fly' and is not saved in the database Typically used to retrieve all actions regarding an object, without needing to specifically track this object

matches(*action*)
Returns true if an action is to be tracked by the Tracker object

update_unread(*already_fetched=()*)
Retrieves the actions having occurred after the last time the tracker was updated and mark them as unread (bulk-add to unread_actions).

4.3.3 DeletedItem

See *Deleted items*.

class actrack.models.**DeletedItem**(**args*, ***kwargs*)
A model to keep track of objects that have been deleted but that still need to be linked by Action instances

ctype
The deleted instance's content type

description
The deleted instance's description when the instance was deleted

serialization
The deleted instance's serialization in JSON when the instance was deleted

Warning: This documentation is a work in progress. Some features may be undocumented, or only lightly documented. It may be necessary to have a look at the source code for more details on some features.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index