django-auditlog Documentation

Release 0.4.5

Jan-Jelle Kester

Nov 04, 2019

Contents

django-auditlog (Auditlog) is a reusable app for Django that makes logging object changes a breeze. Auditlog tries to use as much as Python and Django's built in functionality to keep the list of dependencies as short as possible. Also, Auditlog aims to be fast and simple to use.

Auditlog is created out of the need for a simple Django app that logs changes to models along with the user who made the changes (later referred to as actor). Existing solutions seemed to offer a type of version control, which was found excessive and expensive in terms of database storage and performance.

The core idea of Auditlog is similar to the log from Django's admin. However, Auditlog is much more flexible than the log from Django's admin app (`django.contrib.admin`). Also, Auditlog saves a summary of the changes in JSON format, so changes can be tracked easily.

CHAPTER 1

Contents

1.1 Installation

Installing Auditlog is simple and straightforward. First of all, you need a copy of Auditlog on your system. The easiest way to do this is by using the Python Package Index (PyPI). Simply run the following command:

```
pip install django-auditlog
```

Instead of installing Auditlog via PyPI, you can also clone the Git repository or download the source code via GitHub. The repository can be found at https://github.com/jjkester/django-auditlog/.

Requirements

- Python 2.7, 3.4 or higher
- Django 1.8 or higher

Auditlog is currently tested with Python 2.7 and 3.4 and Django 1.8, 1.9 and 1.10. The latest test report can be found at https://travis-ci.org/jjkester/django-auditlog.

1.1.1 Adding Auditlog to your Django application

To use Auditlog in your application, just add `'auditlog'` to your project's `INSTALLED_APPS` setting and run `manage.py migrate` to create/upgrade the necessary database structure.

If you want Auditlog to automatically set the actor for log entries you also need to enable the middleware by adding `'auditlog.middleware.AuditlogMiddleware'` to your `MIDDLEWARE_CLASSES` setting. Please check *Usage* for more information.

1.2 Usage

1.2.1 Manually logging changes

Auditlog log entries are simple `LogEntry` model instances. This makes creating a new log entry very easy. For even more convenience, `LogEntryManager` provides a number of methods which take some work out of your hands.

See *Internals* for all details.

1.2.2 Automatically logging changes

Auditlog can automatically log changes to objects for you. This functionality is based on Django's signals, but linking your models to Auditlog is even easier than using signals.

Registering your model for logging can be done with a single line of code, as the following example illustrates:

```
from auditlog.registry import auditlog
from django.db import models

class MyModel(models.Model):
    pass
    # Model definition goes here

auditlog.register(MyModel)
```

It is recommended to place the register code (`auditlog.register(MyModel)`) at the bottom of your `models.py` file. This ensures that every time your model is imported it will also be registered to log changes. Auditlog makes sure that each model is only registered once, otherwise duplicate log entries would occur.

Excluding fields

Fields that are excluded will not trigger saving a new log entry and will not show up in the recorded changes.

To exclude specific fields from the log you can pass `include_fields` resp. `exclude_fields` to the `register` method. If `exclude_fields` is specified the fields with the given names will not be included in the generated log entries. If `include_fields` is specified only the fields with the given names will be included in the generated log entries. Explicitly excluding fields through `exclude_fields` takes precedence over specifying which fields to include.

For example, to exclude the field `last_updated`, use:

```
auditlog.register(MyModel, exclude_fields=['last_updated'])
```

New in version 0.3.0: Excluding fields

Mapping fields

If you have field names on your models that aren't intuitive or user friendly you can include a dictionary of field mappings during the *register()* call.

```
class MyModel(modelsModel):
    sku = models.CharField(max_length=20)
    version = models.CharField(max_length=5)
    product = models.CharField(max_length=50, verbose_name='Product Name')
    history = AuditLogHistoryField()

auditlog.register(MyModel, mapping_fields={'sku': 'Product No.', 'version': 'Product␣
↪Revision'})
```

```
log = MyModel.objects.first().history.latest()
log.changes_display_dict
// retrieves changes with keys Product No. Product Revision, and Product Name
// If you don't map a field it will fall back on the verbose_name
```

New in version 0.5.0.

You do not need to map all the fields of the model, any fields not mapped will fall back on their `verbose_name`. Django provides a default `verbose_name` which is a "munged camel case version" so `product_name` would become `Product Name` by default.

1.2.3 Actors

When using automatic logging, the actor is empty by default. However, auditlog can set the actor from the current request automatically. This does not need any custom code, adding a middleware class is enough. When an actor is logged the remote address of that actor will be logged as well.

To enable the automatic logging of the actors, simply add the following to your `MIDDLEWARE_CLASSES` setting in your project's configuration file:

```
MIDDLEWARE_CLASSES = (
    # Request altering middleware, e.g., Django's default middleware classes
    'auditlog.middleware.AuditlogMiddleware',
    # Other middleware
)
```

It is recommended to keep all middleware that alters the request loaded before Auditlog's middleware.

Warning: Please keep in mind that every object change in a request that gets logged automatically will have the current request's user as actor. To only have some object changes to be logged with the current request's user as actor manual logging is required.

1.2.4 Object history

Auditlog ships with a custom field that enables you to easily get the log entries that are relevant to your object. This functionality is built on Django's content types framework (`django.contrib.contenttypes`). Using this field in your models is equally easy as any other field:

```
from auditlog.models import AuditlogHistoryField
from auditlog.registry import auditlog
from django.db import models

class MyModel(models.Model):
    history = AuditlogHistoryField()
    # Model definition goes here

auditlog.register(MyModel)
```

`AuditlogHistoryField` accepts an optional `pk_indexable` parameter, which is either `True` or `False`, this defaults to `True`. If your model has a custom primary key that is not an integer value, `pk_indexable` needs to be set to `False`. Keep in mind that this might slow down queries.

The `AuditlogHistoryField` provides easy access to `LogEntry` instances related to the model instance. Here is an example of how to use it:

```
<div class="table-responsive">
  <table class="table table-striped table-bordered">
    <thead>
      <tr>
        <th>Field</th>
        <th>From</th>
        <th>To</th>
      </tr>
    </thead>
    <tbody>
    {% for key, value in mymodel.history.latest.changes_dict.iteritems %}
      <tr>
        <td>{{ key }}</td>
        <td>{{ value.0|default:"None" }}</td>
        <td>{{ value.1|default:"None" }}</td>
      </tr>
    {% empty %}
      <p>No history for this item has been logged yet.</p>
    {% endfor %}
    </tbody>
  </table>
</div>
```

If you want to display the changes in a more human readable format use the `LogEntry`'s `changes_display_dict` instead. The `changes_display_dict` will make a few cosmetic changes to the data.

- Mapping Fields property will be used to display field names, falling back on `verbose_name` if no mapping field is present
- Fields with a value whose length is greater than 140 will be truncated with an ellipsis appended
- Date, Time, and DateTime fields will follow `L10N` formatting. If `USE_L10N=False` in your settings it will fall back on the settings defaults defined for `DATE_FORMAT`, `TIME_FORMAT`, and `DATETIME_FORMAT`
- Fields with `choices` will be translated into their human readable form, this feature also supports choices defined on `django-multiselectfield` and Postgres's native `ArrayField`

Check out the internals for the full list of attributes you can use to get associated `LogEntry` instances.

1.2.5 Many-to-many relationships

New in version 0.3.0.

> **Warning:** To-many relations are not officially supported. However, this section shows a workaround which can be used for now. In the future, this workaround may be used in an official API or a completly different strategy might be chosen. **Do not rely on the workaround here to be stable across releases.**

By default, many-to-many relationships are not tracked by Auditlog.

The history for a many-to-many relationship without an explicit 'through' model can be recorded by registering this model as follows:

```
auditlog.register(MyModel.related.through)
```

The log entries for all instances of the 'through' model that are related to a `MyModel` instance can be retrieved with the `LogEntryManager.get_for_objects()` method. The resulting QuerySet can be combined with any other queryset of `LogEntry` instances. This way it is possible to get a list of all changes on an object and its related objects:

```
obj = MyModel.objects.first()
rel_history = LogEntry.objects.get_for_objects(obj.related.all())
full_history = (obj.history.all() | rel_history.all()).order_by('-timestamp')
```

1.2.6 Management commands

New in version 0.4.0.

Auditlog provides the `auditlogflush` management command to clear all log entries from the database.

The command asks for confirmation, it is not possible to execute the command without giving any form of (simulated) user input.

Warning: Using the `auditlogflush` command deletes all log entries permanently and irreversibly from the database.

1.2.7 Django Admin integration

New in version 0.4.1.

When `auditlog` is added to your `INSTALLED_APPS` setting a customized admin class is active providing an enhanced Django Admin interface for log entries.

1.3 Internals

You might be interested in the way things work on the inside of Auditlog. This section covers the internal APIs of Auditlog which is very useful when you are looking for more advanced ways to use the application or if you like to contribute to the project.

The documentation below is automatically generated from the source code.

1.3.1 Models and fields

1.3.2 Middleware

1.3.3 Signal receivers

1.3.4 Calculating changes

`auditlog.diff.`**`get_field_value`**(*obj*, *field*)

Gets the value of a given model instance field. :param obj: The model instance. :type obj: Model :param field: The field you want to find the value of. :type field: Any :return: The value of the field as a string. :rtype: str

`auditlog.diff.`**`get_fields_in_model`**(*instance*)

Returns the list of fields in the given model instance. Checks whether to use the official _meta API or use the raw data. This method excludes many to many fields.

Parameters **instance** (*Model*) – The model instance to get the fields for

Returns The list of fields for the given model (instance)

Return type list

auditlog.diff.**model_instance_diff**(*old*, *new*)

Calculates the differences between two model instances. One of the instances may be None (i.e., a newly created model or deleted model). This will cause all fields with a value to have changed (from None).

Parameters

- **old** (*Model*) – The old state of the model instance.
- **new** (*Model*) – The new state of the model instance.

Returns A dictionary with the names of the changed fields as keys and a two tuple of the old and new field values as value.

Return type dict

auditlog.diff.**track_field**(*field*)

Returns whether the given field should be tracked by Auditlog.

Untracked fields are many-to-many relations and relations to the Auditlog LogEntry model.

Parameters **field** (*Field*) – The field to check.

Returns Whether the given field should be tracked.

Return type bool

1.3.5 Registry

CHAPTER 2

Contribute to Auditlog

Note: Due to multiple reasons the development of Auditlog is not a priority for me at this moment. Therefore progress might be slow. This does not mean that this project is abandoned! Community involvement in the form of pull requests is very much appreciated. Also, if you like to take Auditlog to the next level and be a permanent contributor, please contact the author. Contact information can be found via GitHub.

If you discovered a bug or want to improve the code, please submit an issue and/or pull request via GitHub. Before submitting a new issue, please make sure there is no issue submitted that involves the same problem.

GitHub repository: https://github.com/jjkester/django-auditlog
Issues: https://github.com/jjkester/django-auditlog/issues

Python Module Index

a

Index