django-auditlog

Release 3.0.0.post3+ga652eeb

Jan-Jelle Kester and contributors

Apr 29, 2024

CONTENTS

1 Contents 3

1.1 Installation 3

1.2 Usage 3

1.3 Upgrading to version 3 11

1.4 Internals 12

2 Contribute to Auditlog 21

Python Module Index 23

Index 25

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

ONE

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/jazzband/django-auditlog/.

Requirements

- Python 3.8 or higher
- Django 3.2, 4.2 and 5.0

Auditlog is currently tested with Python 3.8+ and Django 3.2, 4.2 and 5.0. The latest test report can be found at https://github.com/jazzband/django-auditlog/actions.

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` 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 django.db import models

from auditlog.registry import auditlog

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.

Logging access

By default, Auditlog will only log changes to your model instances. If you want to log access to your model instances as well, Auditlog provides a mixin class for that purpose. Simply add the `auditlog.mixins.LogAccessMixin` to your class based view and Auditlog will log access to your model instances. The mixin expects your view to have a `get_object` method that returns the model instance for which access shall be logged - this is usually the case for DetailViews and UpdateViews.

A DetailView utilizing the LogAccessMixin could look like the following example:

```
from django.views.generic import DetailView

from auditlog.mixins import LogAccessMixin

class MyModelDetailView(LogAccessMixin, DetailView):
    model = MyModel

    # View code goes here
```

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'])
```

Added 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.

```
from django.db import models

from auditlog.models import AuditlogHistoryField
from auditlog.registry import auditlog

class MyModel(models.Model):
    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
```

Added 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.

Masking fields

Fields that contain sensitive info and we want keep track of field change but not to contain the exact change.

To mask specific fields from the log you can pass `mask_fields` to the `register` method. If `mask_fields` is specified, the first half value of the fields is masked using `*`.

For example, to mask the field `address`, use:

```
auditlog.register(MyModel, mask_fields=['address'])
```

Added in version 2.0.0: Masking fields

Many-to-many fields

Changes to many-to-many fields are not tracked by default. If you want to enable tracking of a many-to-many field on a model, pass `m2m_fields` to the `register` method:

```
auditlog.register(MyModel, m2m_fields={"tags", "contacts"})
```

This functionality is based on the `m2m_changed` signal sent by the `through` model of the relationship.

Note that when the user changes multiple many-to-many fields on the same object through the admin, both adding and removing some objects from each, this code will generate multiple log entries: each log entry will represent a single operation (add or delete) of a single field, e.g. if you both add and delete values from 2 fields on the same form in the same request, you'll get 4 log entries.

Added in version 2.1.0.

Serialized Data

The state of an object following a change action may be optionally serialized and persisted in the `LogEntry.serialized_data` JSONField. To enable this feature for a registered model, add `serialize_data=True` to the kwargs on the `auditlog.register(...)` method. Object serialization will not occur unless this kwarg is set.

```
auditlog.register(MyModel, serialize_data=True)
```

Objects are serialized using the Django core serializer. Keyword arguments may be passed to the serializer through `serialize_kwargs`.

```
auditlog.register(
    MyModel,
    serialize_data=True,
    serialize_kwargs={"fields": ["foo", "bar", "biz", "baz"]}
)
```

Note that all fields on the object will be serialized unless restricted with one or more configurations. The *serialize_kwargs* option contains a *fields* argument and this may be given an inclusive list of field names to serialize (as shown above). Alternatively, one may set `serialize_auditlog_fields_only` to `True` when registering a model with `exclude_fields` and `include_fields` set (as shown below). This will cause the data persisted in `LogEntry.serialized_data` to be limited to the same scope that is persisted within the `LogEntry.changes` field.

```
auditlog.register(
    MyModel,
    exclude_fields=["ssn", "confidential"]
    serialize_data=True,
    serialize_auditlog_fields_only=True
)
```

Field masking is supported in object serialization. Any value belonging to a field whose name is found in the `mask_fields` list will be masked in the serialized object data. Masked values are obfuscated with asterisks in the same way as they are in the `LogEntry.changes` field.

1.2.3 Correlation ID

You can store a correlation ID (cid) in the log entries by:

1. Reading from a request header (specified by *AUDITLOG_CID_HEADER*)
2. Using a custom cid getter (specified by *AUDITLOG_CID_GETTER*)

Using the custom getter is helpful for integrating with a third-party cid package such as django-cid.

1.2.4 Settings

AUDITLOG_INCLUDE_ALL_MODELS

You can use this setting to register all your models:

```
AUDITLOG_INCLUDE_ALL_MODELS=True
```

Added in version 2.1.0.

AUDITLOG_EXCLUDE_TRACKING_FIELDS

You can use this setting to exclude named fields from ALL models. This is useful when lots of models share similar fields like `` `created` `` and `` `modified` `` and you want those excluded from logging. It will be considered when `AUDITLOG_INCLUDE_ALL_MODELS` is *True*.

```
AUDITLOG_EXCLUDE_TRACKING_FIELDS = (
    "created",
    "modified"
)
```

Added in version 3.0.0.

AUDITLOG_EXCLUDE_TRACKING_FIELDS

When using “AuditlogMiddleware”, the IP address is logged by default, you can use this setting to exclude the IP address from logging. It will be considered when `AUDITLOG_DISABLE_REMOTE_ADDR` is *True*.

```
AUDITLOG_DISABLE_REMOTE_ADDR = True
```

Added in version 3.0.0.

AUDITLOG_EXCLUDE_TRACKING_MODELS

You can use this setting to exclude models in registration process. It will be considered when `AUDITLOG_INCLUDE_ALL_MODELS` is *True*.

```
AUDITLOG_EXCLUDE_TRACKING_MODELS = (
    "<app_name>",
    "<app_name>.<model>"
)
```

Added in version 2.1.0.

AUDITLOG_INCLUDE_TRACKING_MODELS

You can use this setting to configure your models registration and other behaviours. It must be a list or tuple. Each item in this setting can be a:

- `str`: To register a model.
- `dict`: To register a model and define its logging behaviour. e.g. include_fields, exclude_fields.

```
AUDITLOG_INCLUDE_TRACKING_MODELS = (
    "<appname>.<model1>",
    {
        "model": "<appname>.<model2>",
        "include_fields": ["field1", "field2"],
        "exclude_fields": ["field3", "field4"],
        "mapping_fields": {
            "field1": "FIELD",
        },
        "mask_fields": ["field5", "field6"],
        "m2m_fields": ["field7", "field8"],
        "serialize_data": True,
        "serialize_auditlog_fields_only": False,
        "serialize_kwargs": {"fields": ["foo", "bar", "biz", "baz"]},
    },
    "<appname>.<model3>",
)
```

Added in version 2.1.0.

AUDITLOG_DISABLE_ON_RAW_SAVE

Disables logging during raw save. (I.e. for instance using loaddata)

Note: M2M operations will still be logged, since they're never considered *raw*. To disable them you must remove their setting or use the *disable_auditlog* context manager.

Added in version 2.2.0.

AUDITLOG_CID_HEADER

The request header containing the Correlation ID value to use in all log entries created as a result of the request. The value can of in the format *HTTP_MY_HEADER* or *my-header*.

Added in version 3.0.0.

AUDITLOG_CID_GETTER

The function to use to retrieve the Correlation ID. The value can be a callable or a string import path.

If the value is *None*, the default getter will be used.

Added in version 3.0.0.

1.2.5 Actors

Middleware

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` setting in your project's configuration file:

```
MIDDLEWARE = (
    # 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.

1.2.6 Context managers

Set actor

To enable the automatic logging of the actors outside of request context (e.g. in a Celery task), you can use a context manager:

```python
from auditlog.context import set_actor

def do_stuff(actor_id: int):
    actor = get_user(actor_id)
    with set_actor(actor):
        # if your code here leads to creation of LogEntry instances, these will have the actor set
        ...
```

Added in version 2.1.0.

Disable auditlog

Disable auditlog temporary, for instance if you need to install a large fixture on a live system or cleanup corrupt data:

```python
from auditlog.context import disable_auditlog

with disable_auditlog():
    # Do things silently here
    ...
```

Added in version 2.2.0.

1.2.7 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:

```python
from django.db import models

from auditlog.models import AuditlogHistoryField
from auditlog.registry import auditlog

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.items %}
      <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.

1.2.8 Many-to-many relationships

Added in version 0.3.0.

Note: This section shows a workaround which can be used to track many-to-many relationships on older versions of django-auditlog. For versions 2.1.0 and onwards, please see the many-to-many fields section of *Automatically logging changes*. **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')
```

1.2.9 Management commands

Added in version 0.4.0.

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

By default, the command asks for confirmation. It is possible to run the command with the -y or --yes flag to skip confirmation and immediately delete all entries.

You may also specify a date using the -b or --before-date option in ISO 8601 format (YYYY-mm-dd) to delete all log entries prior to a given date. This may be used to implement time based retention windows.

Added in version 2.1.0.

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

1.2.10 Django Admin integration

Added 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 Upgrading to version 3

Version 3.0.0 introduces breaking changes. Please review the migration guide below before upgrading. If you're new to django-auditlog, you can ignore this part.

The major change in the version is that we're finally storing changes as json instead of json-text. To convert the existing records, this version has a database migration that does just that. However, this migration will take a long time if you have a huge amount of records, causing your database and application to be out of sync until the migration is complete.

To avoid this, follow these steps:

1. Before upgrading the package, add these two variables to settings.py:
   - AUDITLOG_TWO_STEP_MIGRATION = True
   - AUDITLOG_USE_TEXT_CHANGES_IF_JSON_IS_NOT_PRESENT = True
2. Upgrade the package. Your app will now start storing new records as JSON, but the old records will accessible via LogEntry.changes_text.
3. Use the newly added auditlogmigratejson command to migrate your records. Run django-admin auditlogmigratejson --help to get more information.

4. Once all records are migrated, remove the variables listed above, or set their values to `False`.

1.4 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.

1.4.1 Models and fields

`class auditlog.models.AuditlogHistoryField`(*pk_indexable=True*, *delete_related=False*, ***kwargs*)

A subclass of py:class:*django.contrib.contenttypes.fields.GenericRelation* that sets some default variables. This makes it easier to access Auditlog's log entries, for example in templates.

By default, this field will assume that your primary keys are numeric, simply because this is the most common case. However, if you have a non-integer primary key, you can simply pass `pk_indexable=False` to the constructor, and Auditlog will fall back to using a non-indexed text based field for this model.

Using this field will not automatically register the model for automatic logging. This is done so you can be more flexible with how you use this field.

Parameters

- **pk_indexable** (*bool*) – Whether the primary key for this model is not an `int` or `long`.
- **delete_related** (*bool*) – Delete referenced auditlog entries together with the tracked object. Defaults to False to keep the integrity of the auditlog.

`bulk_related_objects`(*objs*, *using='default'*)

Return all objects related to `objs` via this `GenericRelation`.

`class auditlog.models.LogEntry`(**args*, ***kwargs*)

Represents an entry in the audit log. The content type is saved along with the textual and numeric (if available) primary key, as well as the textual representation of the object when it was saved. It holds the action performed and the fields that were changed in the transaction.

If AuditlogMiddleware is used, the actor will be set automatically. Keep in mind that editing / re-saving LogEntry instances may set the actor to a wrong value - editing LogEntry instances is not recommended (and it should not be necessary).

`class Action`

The actions that Auditlog distinguishes: creating, updating and deleting objects. Viewing objects is not logged. The values of the actions are numeric, a higher integer value means a more intrusive action. This may be useful in some cases when comparing actions because the `__lt`, `__lte`, `__gt`, `__gte` lookup filters can be used in queries.

The valid actions are `Action.CREATE`, `Action.UPDATE`, `Action.DELETE` and `Action.ACCESS`.

`exception DoesNotExist`

`exception MultipleObjectsReturned`

`property changes_dict`

Returns
The changes recorded in this log entry as a dictionary object.

`property changes_display_dict`

Returns
The changes recorded in this log entry intended for display to users as a dictionary object.

`property changes_str`

Return the changes recorded in this log entry as a string. The formatting of the string can be customized by setting alternate values for colon, arrow and separator. If the formatting is still not satisfying, please use *`LogEntry.changes_dict()`* and format the string yourself.

Parameters

- **`colon`** – The string to place between the field name and the values.
- **`arrow`** – The string to place between each old and new value.
- **`separator`** – The string to place between each field.

Returns
A readable string of the changes in this log entry.

`class auditlog.models.LogEntryManager`(**args*, ***kwargs*)

Custom manager for the *`LogEntry`* model.

`get_for_model`(*model*)

Get log entries for all objects of a specified type.

Parameters
`model` (*`class`*) – The model to get log entries for.

Returns
QuerySet of log entries for the given model.

Return type
QuerySet

`get_for_object`(*instance*)

Get log entries for the specified model instance.

Parameters
`instance` (*`Model`*) – The model instance to get log entries for.

Returns
QuerySet of log entries for the given model instance.

Return type
QuerySet

`get_for_objects`(*queryset*)

Get log entries for the objects in the specified queryset.

Parameters
`queryset` (*`QuerySet`*) – The queryset to get the log entries for.

Returns
The LogEntry objects for the objects in the given queryset.

Return type
QuerySet

`log_create`(*instance*, *force_log: bool = False*, ***kwargs*)

Helper method to create a new log entry. This method automatically populates some fields when no explicit value is given.

Parameters

- **instance** (*Model*) – The model instance to log a change for.
- **force_log** (*bool*) – Create a LogEntry even if no changes exist.
- **kwargs** – Field overrides for the *LogEntry* object.

Returns
The new log entry or *None* if there were no changes.

Return type
LogEntry

`log_m2m_changes`(*changed_queryset*, *instance*, *operation*, *field_name*, ***kwargs*)

Create a new "changed" log entry from m2m record.

Parameters

- **changed_queryset** (*QuerySet*) – The added or removed related objects.
- **instance** (*Model*) – The model instance to log a change for.
- **operation** – "add" or "delete".
- **field_name** (*str*) – The name of the changed m2m field.
- **kwargs** – Field overrides for the *LogEntry* object.

Returns
The new log entry or *None* if there were no changes.

Return type
LogEntry

1.4.2 Middleware

`class auditlog.middleware.AuditlogMiddleware`(*get_response=None*)

Middleware to couple the request's user to log items. This is accomplished by currying the signal receiver with the user from the request (or None if the user is not authenticated).

1.4.3 Correlation ID

`auditlog.cid.get_cid`() → str | None

Calls the cid getter function based on *settings.AUDITLOG_CID_GETTER*

If the setting value is:

- None: then it calls the default getter (which retrieves the value set in *set_cid*)
- callable: then it calls the function
- type(str): then it imports the function and then call it

The result is then returned to the caller.

If your custom getter does not depend on *set_header()*, then we recommend setting *settings.AUDITLOG_CID_GETTER* to *None*.

Returns
The correlation ID

auditlog.cid.**set_cid**(*request: HttpRequest | None = None*) → None

A function to read the cid from a request. If the header is not in the request, then we set it to *None*.

Note: we look for the value of *AUDITLOG_CID_HEADER* in *request.headers* and in *request.META*.

This function doesn't do anything if the user is supplying their own *AUDITLOG_CID_GETTER*.

Parameters
request – The request to get the cid from.

Returns
None

1.4.4 Signal receivers

auditlog.receivers.**check_disable**(*signal_handler*)

Decorator that passes along disabled in kwargs if any of the following is true: - 'auditlog_disabled' from thread-local is true - raw = True and AUDITLOG_DISABLE_ON_RAW_SAVE is True

auditlog.receivers.**log_access**(*sender*, *instance*, ***kwargs*)

Signal receiver that creates a log entry when a model instance is accessed in a AccessLogDetailView.

Direct use is discouraged, connect your model through `auditlog.registry.register()` instead.

auditlog.receivers.**log_create**(*sender*, *instance*, *created*, ***kwargs*)

Signal receiver that creates a log entry when a model instance is first saved to the database.

Direct use is discouraged, connect your model through `auditlog.registry.register()` instead.

auditlog.receivers.**log_delete**(*sender*, *instance*, ***kwargs*)

Signal receiver that creates a log entry when a model instance is deleted from the database.

Direct use is discouraged, connect your model through `auditlog.registry.register()` instead.

auditlog.receivers.**log_update**(*sender*, *instance*, ***kwargs*)

Signal receiver that creates a log entry when a model instance is changed and saved to the database.

Direct use is discouraged, connect your model through `auditlog.registry.register()` instead.

auditlog.receivers.**make_log_m2m_changes**(*field_name*)

Return a handler for m2m_changed with field_name enclosed.

1.4.5 Custom Signals

Django Auditlog provides two custom signals that will hook in before and after any Auditlog record is written from a `create`, `update`, `delete`, or `accessed` action on an audited model.

`auditlog.signals.`**`pre_log = <django.dispatch.dispatcher.Signal object>`**

Whenever an audit log entry is written, this signal is sent before writing the log. Keyword arguments sent with this signal:

Parameters

- **`sender`** (*`class`*) – The model class that's being audited.
- **`instance`** (*`Any`*) – The actual instance that's being audited.
- **`action`** (`Action`) – The action on the model resulting in an audit log entry. Type: *`auditlog.models.LogEntry.Action`*

The receivers' return values are sent to any *`post_log()`* signal receivers, with one exception: if any receiver returns False, no logging will be made. This can be useful if logging should be conditionally enabled / disabled

`auditlog.signals.`**`post_log = <django.dispatch.dispatcher.Signal object>`**

Whenever an audit log entry is written, this signal is sent after writing the log. This signal is also fired when there is an error in creating the log.

Keyword arguments sent with this signal:

Parameters

- **`sender`** (*`class`*) – The model class that's being audited.
- **`instance`** (*`Any`*) – The actual instance that's being audited.
- **`action`** (`Action`) – The action on the model resulting in an audit log entry. Type: *`auditlog.models.LogEntry.Action`*
- **`changes`** (*`Optional[dict]`*) – The changes that were logged. If there was en error while determining the changes, this will be None. In some cases, such as when logging access to the instance, the changes will be an empty dict.
- **`log_entry`** (*`Optional[`*`LogEntry`*`]`*) – The log entry that was created and stored in the database. If there was an error, this will be None.
- **`log_created`** (*`bool`*) – Was the log actually created? This could be false if there was an error in creating the log.
- **`error`** (*`Optional[Exception]`*) – The error, if one occurred while saving the audit log entry. `None`, otherwise
- **`pre_log_results`** (*`List[Tuple[method,Any]]`*) – List of tuple pairs `[(pre_log_receiver, pre_log_response)]`, where `pre_log_receiver` is the receiver method, and `pre_log_response` is the corresponding response of that method. If there are no *`pre_log`* receivers, then the list will be empty. `pre_log_receiver` is guaranteed to be non-null, but `pre_log_response` may be `None`. This depends on the corresponding `pre_log_receiver`'s return value.

Added in version 3.0.0.

1.4.6 Calculating changes

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

Gets the value of a given model instance field.

Parameters

- **obj** (*Model*) – The model instance.
- **field** (*Any*) – The field you want to find the value of.

Returns
The value of the field as a string.

Return type
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.**mask_str**(*value: str*) → str

Masks the first half of the input string to remove sensitive data.

Parameters
value (*str*) – The value to mask.

Returns
The masked version of the string.

Return type
str

auditlog.diff.**model_instance_diff**(*old: Model | None*, *new: Model | None*, *fields_to_check=None*)

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.
- **fields_to_check** (*Iterable*) – An iterable of the field names to restrict the diff to, while ignoring the rest of the model's fields. This is used to pass the *update_fields* kwarg from the model's *save* method.

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

1.4.7 Registry

exception `auditlog.registry.AuditLogRegistrationError`

class `auditlog.registry.AuditlogModelRegistry`(*create: bool = True*, *update: bool = True*, *delete: bool = True*, *access: bool = True*, *m2m: bool = True*, *custom: Dict[ModelSignal, Callable] | None = None*)

A registry that keeps track of the models that use Auditlog to track changes.

`contains`(*model: ModelBase*) → bool

Check if a model is registered with auditlog.

Parameters
`model` – The model to check.

Returns
Whether the model has been registered.

Return type
bool

`register`(*model: ModelBase = None*, *include_fields: List[str] | None = None*, *exclude_fields: List[str] | None = None*, *mapping_fields: Dict[str, str] | None = None*, *mask_fields: List[str] | None = None*, *m2m_fields: Collection[str] | None = None*, *serialize_data: bool = False*, *serialize_kwargs: Dict[str, Any] | None = None*, *serialize_auditlog_fields_only: bool = False*)

Register a model with auditlog. Auditlog will then track mutations on this model's instances.

Parameters

- `model` – The model to register.
- `include_fields` – The fields to include. Implicitly excludes all other fields.
- `exclude_fields` – The fields to exclude. Overrides the fields to include.
- `mapping_fields` – Mapping from field names to strings in diff.
- `mask_fields` – The fields to mask for sensitive info.
- `m2m_fields` – The fields to handle as many to many.
- `serialize_data` – Option to include a dictionary of the objects state in the auditlog.
- `serialize_kwargs` – Optional kwargs to pass to Django serializer
- `serialize_auditlog_fields_only` – Only fields being considered in changes will be serialized.

`register_from_settings()`

Register models from settings variables

`unregister`(*model: ModelBase*) → None

Unregister a model with auditlog. This will not affect the database.

Parameters

`model` – The model to unregister.

CHAPTER TWO

CONTRIBUTE TO AUDITLOG

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/jazzband/django-auditlog
Issues: https://github.com/jazzband/django-auditlog/issues

PYTHON MODULE INDEX

a

auditlog.cid, 14
auditlog.diff, 17
auditlog.middleware, 14
auditlog.models, 12
auditlog.receivers, 15
auditlog.registry, 18
auditlog.signals, 16

INDEX

A

auditlog.cid
    module, 14
auditlog.diff
    module, 17
auditlog.middleware
    module, 14
auditlog.models
    module, 12
auditlog.receivers
    module, 15
auditlog.registry
    module, 18
auditlog.signals
    module, 16
AuditlogHistoryField (*class in auditlog.models*), 12
AuditlogMiddleware (*class in auditlog.middleware*), 14
AuditlogModelRegistry (*class in auditlog.registry*), 18
AuditLogRegistrationError, 18

B

bulk_related_objects() (*auditlog.models.AuditlogHistoryField method*), 12

C

changes_dict (*auditlog.models.LogEntry property*), 12
changes_display_dict (*auditlog.models.LogEntry property*), 13
changes_str (*auditlog.models.LogEntry property*), 13
check_disable() (*in module auditlog.receivers*), 15
contains() (*auditlog.registry.AuditlogModelRegistry method*), 18

G

get_cid() (*in module auditlog.cid*), 14
get_field_value() (*in module auditlog.diff*), 17
get_fields_in_model() (*in module auditlog.diff*), 17
get_for_model() (*auditlog.models.LogEntryManager method*), 13
get_for_object() (*auditlog.models.LogEntryManager method*), 13
get_for_objects() (*auditlog.models.LogEntryManager method*), 13

L

log_access() (*in module auditlog.receivers*), 15
log_create() (*auditlog.models.LogEntryManager method*), 14
log_create() (*in module auditlog.receivers*), 15
log_delete() (*in module auditlog.receivers*), 15
log_m2m_changes() (*auditlog.models.LogEntryManager method*), 14
log_update() (*in module auditlog.receivers*), 15
LogEntry (*class in auditlog.models*), 12
LogEntry.Action (*class in auditlog.models*), 12
LogEntry.DoesNotExist, 12
LogEntry.MultipleObjectsReturned, 12
LogEntryManager (*class in auditlog.models*), 13

M

make_log_m2m_changes() (*in module auditlog.receivers*), 15
mask_str() (*in module auditlog.diff*), 17
model_instance_diff() (*in module auditlog.diff*), 17
module
    auditlog.cid, 14
    auditlog.diff, 17
    auditlog.middleware, 14
    auditlog.models, 12
    auditlog.receivers, 15
    auditlog.registry, 18
    auditlog.signals, 16

P

post_log (*in module auditlog.signals*), 16
pre_log (*in module auditlog.signals*), 16

R

register() (*auditlog.registry.AuditlogModelRegistry method*), 18 register_from_settings() (*auditlog.registry.AuditlogModelRegistry method*), 18

S

set_cid() (*in module auditlog.cid*), 15

T

track_field() (*in module auditlog.diff*), 17

U

unregister() (*auditlog.registry.AuditlogModelRegistry method*), 19