Django Audit Events

Sep 14, 2020

Contents:

1 Introduction **3**
- 1.1 Installation . . . 3
- 1.2 Philosophy . . . 3

2 Quick Start **5**
- 2.1 Creating events in views . . . 5
- 2.2 Creating events outside of the view scope . . . 6

3 Configuration **7**

4 Advanced Usage **9**
- 4.1 Common content for multiple events . . . 9
- 4.2 Swapping event model . . . 9
- 4.3 Archiving aged audit events . . . 10

Index **11**

Extensible custom audit events for humans! This Django app allows you to easily create your own events in your project. Currently only works on PostgreSQL.

Let's have a look:

```
def awesome_view(request):
    foo_obj = Foo.objects.get(pk=1)
    # Do something with foo_obj...
    request.audit_context.create_event(
        foo_obj,
        something="done",
        bar="baz"
    )
```

This will create an audit event, including the request URL, logged-in user, remote IP address, and the following event content:

```
>>> event.content
{"something": "done", "bar: "baz"}
```

The content may contain anything that is passed to `create_event` method as keyword arguments, as long as they are JSON serializable.

CHAPTER 1

Introduction

1.1 Installation

You can install this package from PyPI:

```
pip install django-audit-events
```

You need to append it to the `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'django_audit_events',
]
```

And the `MIDDLEWARE`:

```
MIDDLEWARE = [
    ...
    'django_audit_events.middleware.AuditEventsMiddleware',
]
```

Then migrate your project:

```
python manage.py migrate
```

Now you are ready to create your events, take a look at *Quick Start*.

1.2 Philosophy

django-audit-events is the product of the following technical decisions.

- All events must be associated with one content object. Everything else is optional.

- Events should not be created automatically (or magically), because "Explicit is better than implicit".
- Multiple events can be created when processing a single request.
- Events may be created without an HTTP request, such as a Celery task or a management command.
- Developers should not need to check if an audit context exists or not when creating events. That is the reason why `AuditContextMixin` always creates a blank audit context.

CHAPTER 2

Quick Start

This page shows some examples of the basic usage. Audit events are created by the audit context. There are three methods for creating events:

`new_event()` It only creates an event instance. All fields, including the `content_object`, must be set by the developer. Also, after setting fields in the event, it must be saved.

`create_event(content_object, **content)` It creates an event instance and sets the `content_object`. All other keyword arguments are stored in the `content` field of the event. Then the model is saved.

`create_fields_event(content_object, *fields, **content)` It behaves like `create_event` but additionally appends current field values to event content.

2.1 Creating events in views

Since the audit context is available in the request object, events can easily be generated in views:

```
def awesome_view(request, *args, **kwargs):
    foo_obj = Foo.objects.get(pk=1)
    # Do something with foo_obj...
    request.audit_context.create_event(
        foo_obj,
        something="done",
        bar="baz"
    )
```

You can still access the audit context from the request object, even if you have class-based views.

2.2 Creating events outside of the view scope

If you want to create your audit events in your models or other classes, you can either pass audit_context as an argument or use `AuditContextMixin`

Example using argument:

```
def my_function(arg1, arg2, audit_context=AuditContext(), *args, **kwargs):
    foo_obj = Foo.objects.get(pk=1)
    # Do something with foo_obj...
    audit_context.create_event(
        foo_obj,
        something="done",
        bar="baz"
    )
```

Example using mixin:

```
class Foo(models.Model, AuditContextMixin):
    ...

    def my_method(self):
        self.audit_context.create_event(self, ...)
```

You need to provide the audit context from view before running your method in the model:

```
foo_obj = Foo.objects.get(pk=1)
foo_obj.audit_context = request.audit_context
foo_obj.my_method()
```

CHAPTER 3

Configuration

You can configure Django Audit Events by overriding the variables below in your `settings.py`

`AUDIT_EVENT_MODEL` - default: `django_audit_events.AuditEvent` The event model in use, in the form of `"app.Model"`

`AUDIT_EVENT_ARCHIVE_MODEL` - default: `django_audit_events.ArchivedAuditEvent` The archive event model in use, in the form of `"app.Model"`

`AUDIT_INCLUDE_QUERY_PARAMS` - default: `False` Configuration flag to store query params from requests in the audit context and eventually in audit events.

`AUDIT_INCLUDE_POST_DATA` - default: `False` Configuration flag to store post data from requests in the audit context and eventually in audit events.

`AUDIT_MASK_POST_FIELDS` - default: `("password",)` Post data may contain PII or any other sensitive information such as credit card numbers which you may want to avoid storing in your database. Define the names of the fields that contain sensitive information.

CHAPTER 4

Advanced Usage

Django Audit Events app is highly extensible.

4.1 Common content for multiple events

The audit context can store extra data to be present in every event it creates.

```
audit_context = AuditContext()
audit_context.extra_data["foo"] = "bar"
event = audit_context.new_event()
assert event.content["foo"] == "bar"
```

You can even override this data for a single event if you need.

```
audit_context = AuditContext()
audit_context.extra_data["foo"] = "bar"
event = audit_context.create_event(content_object, foo="baz")
assert event.content["foo"] == "baz"
```

4.2 Swapping event model

You can create your own audit event models by extending `django_audit_events.models.AbstractAuditEvent`.

```
class MyEvent(AbstractAuditEvent):
    ...

    class Meta(AbstractAuditEvent.Meta):
        swappable = "AUDIT_EVENT_MODEL"
```

4.3 Archiving aged audit events

Old audit events may be archived in order to prevent database from being overloaded.

```
from celery.schedules import crontab

CELERYBEAT_SCHEDULE = {
    "archive-old-audit-events": {
        "task": "django_audit_events.tasks.archive_old_audit_events",
        "schedule": crontab(hour=0, minute=0),
        "args": (),
        "kwargs": {"older_than": 90}
    },
}
```

Using `CELERYBEAT_SCHEDULE` configuration like one above, audit events older than 90 days will be archived every day.

Index

A

AUDIT_EVENT_ARCHIVE_MODEL - default:
    django_audit_events.ArchivedAuditEvent,
    7
AUDIT_EVENT_MODEL - default:
    django_audit_events.AuditEvent, 7
AUDIT_INCLUDE_POST_DATA - default:
    False, 7
AUDIT_INCLUDE_QUERY_PARAMS - default:
    False, 7
AUDIT_MASK_POST_FIELDS - default: ("password",), 7

C

create_event(content_object,
    **content), 5
create_fields_event(content_object,
    *fields, **content), 5

N

new_event(), 5