Django Timeline Logger Documentation

Release 1.1.2

Maykin Media

May 30, 2018

Contents

1 Overview **3**

2 Requirements **5**

3 Contents **7**
- 3.1 Installation . . . 7
- 3.2 Usage . . . 7
- 3.3 Sending log reports . . . 10
- 3.4 Settings . . . 11
- 3.5 Contributing . . . 11
- 3.6 Changelog . . . 12

4 License **13**

5 Source Code and contributing **15**

6 Indices and tables **17**

A reusable Django app to log actions and display them in a timeline.

CHAPTER 1

Overview

Django Timeline Logger is a simple pluggable application that adds events logging and reporting to your Django projects.

It easily enables you to generate customized log messages on events, thus providing your backend with a logging system slightly more advanced and customizable than the builtin "admin logs" generated via `LogEntry`.

CHAPTER 2

Requirements

Django Timeline Logger makes use of Django's `contrib.postgres.JSONField`, then your backend will need:

- At least Django-1.11.
- At least PostgreSQL-9.4.
- At least psycopg2-2.5.4.

CHAPTER 3

Contents

3.1 Installation

To install Django Timeline Logger you can use PyPI:

```
pip install django-timeline-logger
```

Once installation is complete, you can enable the application in your Django project by adding it to `INSTALLED_APPS` in the regular way:

```
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    ...

    'timeline_logger',
]
```

Then, run the migrations:

```
python manage.py migrate timeline_logger
```

You can now start using the application. Go to *Usage* section for the details.

3.2 Usage

Django Timeline Logger works by using a custom model `TimelineLog`, which is designed to store:

- A Django model instance (a database object).
- A timestamp.
- A user instance (optional).

- A path to a template (optional, defaults to `timeline_logger/default.txt`).
- A context (optional).

Given those details, it's pretty clear how it works: whenever you want to log an event in your system, you create a `TimelineLog` for it, passing the data you consider useful in the context and using a template to render the message.

The context is stored in a `django.contrib.postgres.JSONField`, which basically accepts a Python dictionary representing JSON data, to be built by you with the data you want to pass to the message template.

3.2.1 Default example

An example of a default usage of the `TimelineLog` model could be as follows. Imagine you have a "blog" application where your users can create posts, stored by using a Django model `Post`.

Using the default behaviour of `TimelineLog`, you can create a log each time a user posts a new entry in the blog, like this:

```
from timeline_logger.models import TimelineLog


# Whatever logic you have before creating the post entry.
...

post = Post.objects.create(
    title='New blog entry',
    body=post_text
)

log = TimelineLog.objects.create(
    content_object=post,
    user=my_user
)
```

There you go. A new timeline entry has been created to record the event that a user posted a new blog entry.

You can then see the default message for such event by calling the `TimelineLog.get_message()` method:

```
log.get_message()
```

That function will return a text string, like this one:

```
'July 1, 2016, 9:08 a.m. - Anonymous user event on New blog entry\n'
```

With all this in place, you can now access the view included with the Django Timeline Logger package: http://localhost:8000/timeline to see a paginated list of your event logs.

3.2.2 Custom messages using templates and context

Of course, you want to have your own custom messages for each different event you want to track, maybe showing in the log as more data as possible. You can easily do that by using regular Django templates in your `TimelineLog` instances.

Let's see an example of usage. Imagine you have a web shop and you want to log user purchases of certain items, whatever the item is. In this scenario, you could have a view to handle a user form submission representing a purchase. You should be able to log each purchase with any details you want by doing something like this:

```
from django.views.generic import CreateView

from timeline_logger.models import TimelineLog

from my_app.models import Invoice, Item


class PurchaseView(CreateView):
    """ Manages a client purchase and creates the invoice """
    model = Invoice
    ...

    def post(self, request, *args, **kwargs):
        response = super(PurchaseView, self).post(request, *args, **kwargs)

        # The sold item.
        item = Item.objects.get(pk=kwargs['item'])

        # Add some extra data to the log message.
        extra_data = {'invoice': self.object}

        # Log the purchase event.
        TimelineLog.objects.create(
            content_object=item,
            user=request.user,
            template='timeline_logger/purchase.txt',
            extra_data=**extra_data
        )

        return response
```

You logged there the "purchase event", passing the `request` object, using a custom template to render your own message and some context for it. A simple template you can write in your `my_app/templates/timeline_logger` directory could look like this:

```
{% load i18n %}
{% blocktrans trimmed with timestamp=log.timestamp user=log.user|default:_('Anonymous␣
↪user') object=log.content_object extra=log.exta_data|safe %}
   {{ timestamp }} - {{ user }} purchased item "{{ object }}", using payment method "{
↪{ extra.invoice.method }}", for a total price of {{ extra.invoice.total }} €.
{% endblocktrans %}
```

So, in your http://localhost:8000/timeline view, this log entry will appear more or less as follows:

> July 4, 2016, 8:13 a.m. - John Doe purchased item "Nescafé Dolce Gusto", using payment method "PayPal", for a total price of 35 €.

3.2.3 Log from requests

Probably you'll better like to log events based on user requests, like for example a user comment in a blog post, a form submission, a click in a "like" button or a purchase in your web shop.

You can easily do so by using the `TimelineLog.log_from_request` method, which accepts a Django `HTTPRequest` object (accessible in all Django views via the `request` parameter or the `self.request` view class attribute) and a Django model instance, plus an optional template and its context.

In our previous example, we can substitute the `TimelineLog.objects.create(...` part by this:

```
TimelineLog.log_from_request(
    request,
    item,
    'timeline_logger/purchase.txt',
    **extra_data
)
```

And the resulting log instance and message will be the same.

3.2.4 Django-import-export integration

Django-timeline-logger ships with a `ModelResource`:

```
   from timeline_logger.resources import TimelineLogResource

   ...

It's not enabled in the default admin, as django-import-export is an
optional dependency.
```

3.3 Sending log reports

Timeline Logger includes a Django management command that you can add to a cronjob, or trigger manually when you want, to send reports via email about your site usage to those people you want.

3.3.1 Report mailing

The management command can be called like this in "default" mode:

```
python manage.py report_mailing
```

In this mode, **only** those Django system users marked as "staff" member **and** "superuser" will be notified via email. You can change this default behaviour by using some command arguments and Django project *Settings*.

Options

The command options are:

- `--all`: Send notification emails to **all** users registered in the system.
- `--staff`: Send notification emails **only** to the system users marked as `is_staff=True`.
- `--recipients_from_setting`: Send notification emails to those email addresses listed in `TIMELINE_DIGEST_EMAIL_RECIPIENTS` setting.

Custom email notifications

In case you don't like the default look and feel of the HTML notification email, you can design your own template and place it in your project `templates/timeline_logger/` directory using the name `notifications.html`.

3.4 Settings

Timeline Logger default behaviour can also be customized by using the next application settings in your project:

- `TIMELINE_DEFAULT_TEMPLATE`: defines the Django template used for the standard logs message. Defaults to `timeline_logger/default.txt`.
- `TIMELINE_DIGEST_EMAIL_RECIPIENTS`: defines a fixed list of email addresses that must be notified when running the `report_mailing` command, without any else being notified at all. Could be system registered users or not. Defaults to `None`.
- `TIMELINE_DIGEST_EMAIL_SUBJECT`: a string defining a subject for the notification email. Defaults to "Events timeline".
- `TIMELINE_DIGEST_FROM_EMAIL`: the "sender" email that will be used to send the notifications. Defaults to `None`, and then if it's not set it will use Django's `DEFAULT_FROM_EMAIL` setting "webmaster@localhost".
- `TIMELINE_PAGINATE_BY`: the number of log entries that will be shown for each page in your http://localhost:8000/timeline view. Defaults to 25.
- `TIMELINE_USER_EMAIL_FIELD`: in case you are using a custom Django `User` model with the user email stored in a specific field, it allows you to specify such field. Defaults to `'email'`.

3.5 Contributing

To get up and running quickly, fork the github repository and make all your changes in your local clone.

Git-flow is prefered as git workflow, but as long as you make pull requests against the `develop` branch, all should be well. Pull requests should always have tests, and if relevant, documentation updates.

Feel free to create unfinished pull-requests to get the tests to build and get work going, someone else might always want to pick up the tests and/or documentation.

3.5.1 Testing

Django's testcases are used to run the tests.

To run the tests in your (virtual) environment, simple execute

```
python setup.py test
```

This will run the tests with the current python version and Django version installed in your virtual environment.

To run the tests on all supported python/Django versions, use tox.

```
pip install tox
tox
```

If you want to speed this up, you can also use detox. This library will run as much in parallel as possible.

3.5.2 Documentation

The documentation is built with Sphinx. Run *make* to build the documentation:

```
cd docs/
make html
```

You can now open *_build/index.html*.

3.5.3 Coding style

Please stick to PEP8, and use pylint or similar tools to check the code style. Also sort your imports, you may use `isort` for this. In general, we adhere to Django's coding style.

3.6 Changelog

3.6.1 1.1.2 (2018-05-30)

Fixed packaging mistake - Dutch translations are now included.

3.6.2 1.1.1 (2018-05-04)

Added Dutch translations (PR#14, thanks @josvromans)

3.6.3 1.1 (2018-04-17)

- Added django-import-export support
- Added a demo project to showcase the usage/integrations.

3.6.4 1.0 (2018-03-15)

Breaking changes

- Changed the GFK object_id field to a text field, so that non-integer primary keys are supported. This may come at a (small) performance hit. Depending on the data you're storing, the backwards migration may break.
- Dropped Django 1.10 support, per Django's version support policy. If you're still on Django 1.10, you should upgrade, but other than that the app probably still works.

New features

- You can now create log entries without referring to a specific object (thanks @tsiaGeorge).
- Support non-integer PKs for objects (see #7, thanks for the feedback @holms)

Other

- first pass at better supporting internationalization
- cleaned up package a bit, added isort etc.

3.6.5 Pre 1.0

Best guess is looking at the git log, sorry.

CHAPTER 4

License

Licensed under the MIT License.

CHAPTER 5

Source Code and contributing

The source code can be found on Github.

Bugs can also be reported on the Github repository, and pull requests are welcome. See *Contributing* for more details.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search