django-debug-logging Documentation

Release 0.5a1.dev1

Lincoln Loop

August 04, 2015

Contents

Django Debug Logging is a "plugin" for the Django Debug Toolbar that allows users to log the debug toolbar statistics to the database during a site crawl. This allows users to create performance testing plans to exercise the site, and then review and aggregate the results afterwards to identify performance problems.

It also provides a basic UI for browsing the details that have been logged to the database and reviewing aggregated information about test runs. The UI borrows a lot from the custom Sphinx theme by the Read the Docs team, and the Sentry project from Disqus.

The overall goal is to use this tool to monitor performance statistics over time, so that you can see trends and spikes in the number of queries, cache misses, cpu time, etc., and identify where in the app the problems are coming from. It is not intended as a load testing tool, so features like concurrency and warmup periods will not be part of the initial focus.

Contents:

CHAPTER 1

Installation

1.1 Prerequisites

These requirements are installed automatically by *pip* and *easy_install*, and are in the included *requirements.pip* file.

Django Debug Toolbar - This project is designed to work alongside the Django Debug Toolbar and extend its functionality to support logging.

Picklefield - Used to saved pickled versions of settings, sql queries, and cache calls to the database.

1.2 Installing

Before you begin, make sure Django Debug Toolbar is configured and working properly.

Install the project with pip:

```
$ pip install django-debug-logging
```

This should install django-picklefield as well, which is needed.

Next, you'll add *debug_logging* to your INSTALLED_APPS:

```
INSTALLED_APPS = (
    ...
    'debug_toolbar',
    'debug_logging',
)
```

Now, you'll need to replace the standard DebugToolbarMiddleware with a middleware that extends it to add logging functionality. The toolbar will still function normally when logging is disabled.

From your MIDDLEWARE_CLASSES setting, remove:

```
'debug_toolbar.middleware.DebugToolbarMiddleware',
```

Replace it with:

```
'debug_logging.middleware.DebugLoggingMiddleware',
```

Now, you'll need to replace a few of the panels with extended versions that support logging. If you don't want the data from any one of these panels to be logged, you can skip it.

From your DEBUG_TOOLBAR_PANELS setting, remove:

```
'debug_toolbar.panels.cache.CacheDebugPanel',
'debug_toolbar.panels.settings_vars.SettingsVarsDebugPanel',
'debug_toolbar.panels.sql.SQLDebugPanel',
'debug_toolbar.panels.timer.TimerDebugPanel',
```

Replace them with:

```
'debug_logging.panels.cache.CacheLoggingPanel',
'debug_logging.panels.settings_vars.SettingsVarsLoggingPanel',
'debug_logging.panels.sql.SQLLoggingPanel',
'debug_logging.panels.timer.TimerLoggingPanel',
```

There are also a couple of panels that are unique to Django Debug Logging that you may find convenient when logging data over time. If you'd like, you can add them to your DEBUG_TOOLBAR_PANELS setting:

```
'debug_logging.panels.revision.RevisionLoggingPanel',
'debug_logging.panels.identity.IdentityLoggingPanel',
```

Add the debug logging urls to your urls.py:

```
urlpatterns = patterns('',
    ...
    url(r'^debug-logging/', include('debug_logging.urls')),
)
```

The Debug Logger will ignore requests made to this frontend interface, so your log won't be clogged with information you have no use for.

Finally, run syncdb to create the models for statistic logging:

```
$ python manage.py syncdb
```

South migrations are included in case migrations are needed when upgrading to new versions.

1.3 Logging

Requests are logged when they contain a 'DJANGO_DEBUG_LOGGING' header set to True. This header is added automatically by the 'log_urls' command when it is used. To prevent any performance impact from the rendering of the Debug Toolbar, it is not shown when this header is present.

For the best results, don't use the site while a test run is in progress.

CHAPTER 2

Settings

- SQL_EXTRA: This setting determines whether the full details of each query are logged, or just the number of queries and the total time. It defaults to False.
- CACHE_EXTRA: This determines whether the full details of each cache call are logged, or just the summary details. It defaults to '' False''.
- BLACKLIST: Add a list of url prefixes that you would like to exclude from logging here. The url for the Debug Logging frontend interface is added to this blacklist automatically.

CHAPTER 3

Running a Url Test

A management command is included that uses the test client to hit a list of urls in sequence, allowing them to be logged to the database. To use it, first create a list of urls with a new url on each line. Lines beginning with # are ignored.

```
# Main urls
/
/my/url/
/my/other/url/
# Comments
/my/comment/url/
```

Then, enable logging and run the *log_urls* management command:

```
$ python manage.py log_urls myapp/my_urls.txt
```

Unless it is run with a verbosity of 0 the command will output status messages, such as urls that return codes other than 200 and urls that raise errors.

To run the test as an authenticated user, use the username and password options:

```
$ python manage.py log_urls my_urls.txt --username Legen --password dary
```

You can also add a name and a description to your run, if you'd like:

```
$ python manage.py log_urls my_urls.txt --name "Admin Urls" --description "Urls used by site admins"
```

If you'd like to conduct a test run with a tool other than the log_urls management command, you can use the command to manually start and end TestRun objects, so that your results will be organized correctly in the UI. Before you conduct your test, simply run:

```
$ python manage.py log_urls --manual-start
```

Then, when you are finished hitting your desired urls:

```
$ python manage.py log_urls --manual-end
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search