
django-query-profiler

Release 0.0.1

Jun 08, 2022

Installation and configuration

1	installation	3
2	configuration instructions	5
3	choosing profiler level	9
4	customizing the defaults	11
5	how the profiler works	13
6	running tests	17
7	columns in chrome plugin	19

Django query profiler is a profiler for Django applications, for helping developers answer the question “My Django code or page or API is slow, How do I find out why?”

Below are some of the features of the profiler:

1. Shows code paths making N+1 sql calls: Shows the sql with stack_trace which is making N+1 calls, along with sql count
2. Shows the proposed solution: If the solution to reduce sql is to simply apply a select_related or a prefetch_related, this is highlighted as a suggestion
3. Shows exact sql duplicates: Count of the queries where (sql, parameters) is exactly the same. This is the kind of sql where implementing a query cache would help
4. Flame Graph visualisation: Collects all the stack traces together to allow quickly identifying which area(s) of code is producing the load
5. Command line or chrome plugin: The profiler can be called from command line via context manager, or can be invoked via a middleware, and output shown in a chrome plugin
6. Super easy to configure in any application: The only changes are in settings.py file and in urls.py file

To get up and running quickly, *install django-query-profiler*, then read *configuration*, which describes the steps for configuring the profiler in your application

1.1 requirements

This works with any version of django ≥ 2.0 , and running on python ≥ 3.6

1.2 for usage

Simplest way to get the profiler is to use pip, and installing chrome extension from chrome store:

- Python package:

```
$ pip install django-query-profiler
```

You can verify that the application is available on your PYTHONPATH by opening a python interpreter and entering the following commands:

```
>>> import django_query_profiler
>>> django_query_profiler.VERSION
```

- Chrome extension: Download from [chrome webstore](#)

Note that the chrome extension works on any chromium based browser. We have tested it on Google Chrome and Brave Browser

1.3 for development

- clone the git repository for python package from GitHub and run setup.py:

```
$ git clone git://github.com/django-query-profiler/django-query-profiler.git
$ <venv activate command>
$ cd django-query-profiler
$ python setup.py test; python setup.py install;
```

- clone the git repository for chrome plugin and add it to any chromium based browser:

```
git clone git://github.com/django-query-profiler/django-query-profiler-chrome-
↪plugin.git
Open chrome://extensions (command works in any chromium based browser)
- check Developer mode,
- click on load unpacked.
- Select the cloned package above
```


2.1 as chrome plugin

The only places where we will need to change are the settings.py and urls.py file.

settings.py:

```
from django_query_profiler.settings import *

INSTALLED_APPS = (
    ...
    'django_query_profiler',
    ...
)

MIDDLEWARE = (
    ...
    # Request and all middleware that come after our middleware, would be profiled
    'django_query_profiler.client.middleware.QueryProfilerMiddleware',
    ...
)

DATABASES = (
    ...
    # Adding django_query_profiler as a prefix to your ENGINE setting
    # Assuming old ENGINE was "django.db.backends.sqlite3", this would be the new one
    "ENGINE": "django_query_profiler.django.db.backends.sqlite3",
)
```

This works for all the databases supported by Django. In all the databases, the ENGINE settings has to be prepended by *django_query_profiler* for the profiler to work

In case of mysql/mariadb, the ENGINE setting would look like:

django-query-profiler, Release 0.0.1

```
# from "django.db.backends.mysql"
"ENGINE": "django_query_profiler.django.db.backends.mysql",
```

In case of postgres, the ENGINE setting would look like:

```
# from "django.db.backends.postgresql_psycopg2"
"ENGINE": "django_query_profiler.django.db.backends.postgresql_psycopg2",
```

In case of oracle, the ENGINE setting would look like:

```
# from "django.db.backends.oracle"
"ENGINE": "django_query_profiler.django.db.backends.oracle",
```

urls.py:

```
# Add this line to existing urls.py
path('django_query_profiler/', include('django_query_profiler.client.urls'))
```

See this [PR](#) on how to configure this in your application, and how the plugin is going to look like after your configuration



Order is in process. You are in queue: 3

Order summary:

- Order ID: 62
- Name: test
- Order: cappuccino
- Your order placed at: 1:18:49 PM

Please don't refresh the page

The screenshot shows the Chrome DevTools Performance tab with the Django Query Profiler extension active. The interface includes a "Clear Data" button and an "Export to excel" button. The main data is presented in a table titled "Django query profiler data".

Api name	Total request time (in ms)	Server request time (in ms)	Profiler time added (in ms)	Query time (in ms)	Select	Insert	Update	Delete	Transactional sqls	Other sqls	DB Rows fetched	Potential N+1's	Exact sql duplicates	Details link
/accounts/signup/	39	27	0	0	0	0	0	0	0	0	0	0	0	query signature
/accounts/signup/	140	124	1.732	11.713	4	3	2	0	5	0	-	0	5	query signature
/food/order/	20	16	0.496	0.736	4	0	0	0	0	0	-	0	0	query signature
/food/orderlist/	29	5	0.422	0.543	3	0	0	0	0	0	-	0	0	query signature
/food/business/	17	8	0.665	0.824	5	0	0	0	0	0	-	0	0	query signature
/food/orderlist/	16	11	0.369	1.775	3	0	0	0	0	0	-	0	0	query signature
/food/business/	18	14	0.626	3.99	5	0	0	0	0	0	-	0	0	query signature
/food/orderlist/	16	10	0.432	1.578	3	0	0	0	0	0	-	0	0	query signature
/food/business/	20	14	0.703	4.553	5	0	0	0	0	0	-	0	0	query signature

2.2 as chrome plugin without detailed view

We use redis to store the pickled data of *detailed view*, that gets shown when clicking on the “Details Link” in the chrome extension. If redis is not available, we would not be able to see the detailed view, but we can still see the summary view.

In that scenario, the only change would be in the application’s settings.py. We don’t need to add `django_query_profiler` to `INSTALLED_APPS`, and we don’t need to add detailed view url to `urls.py`

settings.py:

```
from django_query_profiler.settings import *

MIDDLEWARE = (
```

(continues on next page)

(continued from previous page)

```

...
# Request and all middleware that come after our middleware, would be profiled
'django_query_profiler.client.middleware.QueryProfilerMiddleware',
...
)

DATABASES = (
...
# Adding django_query_profiler as a prefix to your ENGINE setting
# Assuming old ENGINE was "django.db.backends.sqlite3", this would be the new one
"ENGINE": "django_query_profiler.django.db.backends.sqlite3",
)

```

See this PR on how to configure this in your application, and how the plugin is going to look like after your configuration



Order is in process. You are in queue: 3

Order summary:

- Order ID: 62
- Name: test
- Order: cappuccino
- Your order placed at: 1:18:49 PM

Please don't refresh the page

Api name	Total request time (in ms)	Server request time (in ms)	Profiler time added (in ms)	Query time (in ms)	Select	Insert	Update	Delete	Transactional sqls	Other sqls	DB Rows fetched	Potential N+1's	Exact sql duplicates	Details link
/food/orderlist/	12	6	0.407	1.944	3	0	0	0	0	0	-	0	0	redis_or_urls.py_not_setup
/food/business/	23	16	0.902	1.876	5	0	0	0	0	0	-	0	0	redis_or_urls.py_not_setup
/food/orderlist/	14	8	0.327	1.937	3	0	0	0	0	0	-	0	0	redis_or_urls.py_not_setup
/food/business/	22	15	0.845	2.514	5	0	0	0	0	0	-	0	0	redis_or_urls.py_not_setup

2.3 as context manager

This is helpful if you want to test things out on a command line - it requires only one change to settings.py

settings.py:

```

from django_query_profiler.settings import *

DATABASES = (
...
# Adding django_query_profiler as a prefix to your ENGINE setting
# Assuming old ENGINE was "django.db.backends.sqlite3", this would be the new one
"ENGINE": "django_query_profiler.django.db.backends.sqlite3",
)

```

See this PR on how to configure this in your application,

See this file in the PR to see how to use the context manager And see how easy it is to spot performance issues :-)

3.1 Explaining profiling levels

There are two levels of query profiler, and both of them have different capabilities, and different overheads. The idea to have two different levels is to allow the application developer to choose the right level, based on how much overhead is acceptable for their API.

Currently, there are two levels of profiling:

1. **QUERY_SIGNATURE**: This is the mode where we capture the query as well as the stack-trace. The grouping unit here is (stack-trace, normalized sql), and that grouping helps us to figure out if there are N+1 code paths. Django stack-trace helps us find recommendations - like if a particular query can be stopped by applying `select_related/prefetch_related`

The main overhead comes from calculating stack-traces whenever a query is executed, and for normalizing sql by applying a regex. From our experience of using it in production, the overhead is generally to the order of **1 millisecond per 7 queries**

2. **QUERY**: This is the mode where we just capture queries, and *not* the stack-trace. The grouping unit here is just (sql). Because we don't have access to stack-traces, we don't know if any code path is N+1 or not. We don't have any code recommendation either.

From our experience of using it, the overhead is generally to the order of **1 millisecond per 25 queries**

3.2 Configuring profiling levels

Based on the overhead that profiler adds for every query, the profiler level can be configured in the application settings.py file.

1. The default setting is to run the profiler in *QUERY_SIGNATURE* level. If you want to run the application in *QUERY* level, this is how you should configure in your settings.py file:

```
from django_query_profiler.settings import *
from django_query_profiler.query_profiler_storage import QueryProfilerLevel

def DJANGO_QUERY_PROFILER_LEVEL_FUNC(request) -> Optional[QueryProfilerLevel]:
    return QueryProfilerLevel.QUERY
```

2. If you want to configure it per request, the profiler provides a hook for changing the profiler type given a request object

The profiling level of each API is calculated per request, and can be configured easily. See *customizing_defaults* on how this can be done

customizing the defaults

This doc covers how to change the *django-query-profiler* settings to suit your needs. The attributes that the profiler expects are in `django_query_profiler/settings.py` file

Irrespective of if you are using the chrome plugin, or the context manager, there is one line you would have added to your application settings.py file, for configuring the profiler:

```
from django_query_profiler.settings import *
```

If you want to change any of the defaults, you can import the file, and then define the same parameter again, but with a different value

Here are some example of customizations that can be done:

- If redis is running on a different port, this is what you would have to do in the application settings.py:

```
from django_query_profiler.settings import *
DJANGO_QUERY_PROFILER_REDIS_PORT: int = 8080
```

- If we want to configure the profiler to be run on *certain* api's only. The default setting is to run it on all api's. The way to change the defaults would be do something in the application settings.py:

```
from django_query_profiler.settings import *
from django_query_profiler.query_signature import QueryProfilerLevel

def DJANGO_QUERY_PROFILER_LEVEL_FUNC(request):
    return QueryProfilerLevel.QUERY_SIGNATURE if request.path_info == '/pizza/order'
    ↪ else None
```

- A similar example is if you want the profiler to be run only when the request is coming from internal IPs. Django request META contains the ipaddress, and that can be used to filter out only internal IP address where the profiler would be enabled
- If we want to do say, log the query profiled data to a log file. The way to do it would be:

```
from django_query_profiler.settings import *
from django.http.response import HttpResponseRedirectBase
from django.http import HttpRequest
from django_query_profiler.query_profiler_storage import QueryProfiledData

def DJANGO_QUERY_PROFILER_POST_PROCESSOR(
    query_profiled_data: QueryProfiledData,
    request: HttpRequest,
    response: HttpResponseRedirectBase) -> None:
    logger.info(query_profiled_data)
```

- If we want to remove a particular module from coming in the stack-trace, that gets shown in the detailed view:

```
from django_query_profiler.settings import *

DJANGO_QUERY_PROFILER_APP_MODULES_TO_EXCLUDE += ('package1', 'package2')
```

- An extreme example of customizations - one could write a new chrome plugin &/or your own middleware as well. All *QueryProfilerMiddleware* is doing is to call the context manager, and set some headers which the chrome plugin interprets and append in its table.
 - Rolling out your own chrome plugin & the middleware, which calls the context manager is definitely doable.
 - The *DJANGO_QUERY_PROFILER_POST_PROCESSOR* function is called with the request, response & *QueryProfiledData* - before the response is sent back. It can be used to set additional attributes on the response headers
 - Your application can set other attributes on the response, and your chrome plugin can read those attributes
 - If it is something others can also use, please consider sending a PR :-)

how the profiler works

5.1 Basic Ideas

1. A N+1 query is one which runs the same query (with different params) for the *N related objects* - one for each N - in a loop
 - The best way to know if a *code path* is making N+1 calls is to capture the stack trace & the query, and see if the same stack trace & query combination appears again.
 - We can consider a (stack trace, query) as an aggregate, and maintain a map of this aggregate to count
 - We should segregate the stack trace into (application, django) stack trace. An application stack trace is useful to be displayed, while a django stack trace is not
 - Can we do anything useful with the django stack trace? Does it give us any useful insights - if not, we should discard it (Surprise: It does :-)
 - Where do we store this data? Maybe a thread-local? Lets call this *data collector* module
2. For us to capture stack trace & the query, we have to hook into django to call our *data collector* when a query is executed
 - If we can hook into django to call our data collector when it executes any query, we can also collect other interesting properties, like *exact sql duplicates*, *row count*, and *time taken to run a query*
 - Does Django has hooks for us to execute some function when a query is executed?
3. Once we have the above two pieces figured out, we have to start collecting this data when a request happens, and stop when the request gets finished, ie. figure out the boundaries of profiling
 - A middleware seems like a good boundary, but that would limit us to just requests.
 - A context manager seems like a more generic boundary, and a django middleware can then just call the context manager. This would allow us to use the profiler from command line
4. Once we have this data, where should the data be displayed about the stack trace & the query
 - If it was called as part of context manager, user would know what to do with the data

- If it was called as part of a request, chrome plugin seems like a good place for displaying this data. Middleware can set the data in the headers, and the chrome plugin should be able to read that, and display it in the plugin

5.2 Implementation details

This part is divided by the package that answers the four question/idea discussed above

5.2.1 1. query_profiler_storage

- [github link](#)
- This package has a `data_collector` module where we define a thread-local which exposes three functions:
 - `enter_profiler_mode`: Just sets the profiler to on state
 - `exit_profiler_mode`: Turns off the profiler, and return the profiled data that has been collected since the start of enclosing start block
 - `add_query_profiled_data`: If the profiler is on, start collecting data in its thread-local
- We have defined our data models in the `__init__.py` file. All the bookkeeping code happens in these models, in the python magic functions like the `__add__` ones.
- For capturing the stack trace, and grouping them into (application, django) stack trace, we have `stack_tracer` module
- We are trying to use the django stack trace to figure out if the query is happening because of a forward or a reverse relationship, which helps us to know if this could have been avoided by a `select_related/prefetch_related`.

This is happening in the `django_stack_trace_analyze` module. We are trying to analyze django stack trace, and see if we can find some useful known pattern

5.2.2 2. django

- [github link](#)
- To get a hook from django when it executes a query, that part is done in the `django` module. We are using the fact that django provides a way for us to pass a `DATABASE[ENGINE]` in the settings.py file, as a string.

There are many open source projects which use this hook provided by django, to add some features when connecting to databases:

- `django-postgres-readonly`
- `django-postgrespool`
- `django-sqlserver`
- `custom database backends`

All the above packages have the same part about the `ENGINE` setting - the package has a `base.py` and `__init__.py` file. Looks like, this requirement is coming from `django code`.

- To hook into the django query execution model, all the database in django have a common `CursorWrapper` implementation. This cursor is the last point where we have python/django code. After this layer, the code is handed to the various database drivers

We change the cursorWrapper to our implementation in the module `cursor_wrapper_instrumentation.py`. We use a mixin module `database_wrapper_mixin.py` to do it once for all database, and configure this mixin for each database

In case you are interested to learn about various layers in django, see [this](#) amazing talk by James Bennett. Watch it even if you don't use the profiler :-)

5.2.3 3. client

- [github link](#)
- In the above two modules, we already have all the machinery for the profiler. The one thing that is remaining is to set the boundaries of the profiler - by calling the `enter_profiler_mode` and `exit_profiler_mode` functions. That is exactly what the context manager does.
- The middleware module just calls the context manager, and sets the headers which the chrome plugin expects

5.2.4 4. chrome plugin

- [github link](#)
- This is a different project in the repo. All it does, is see if the headers in the request have the headers which the django query profiler sets. If it has, it parses the response, and add it to table in its devtools panel

CHAPTER 6

running tests

Running test is as simple as running this command:

```
python setup.py test
```

This would run the tests against a sqlite database

If your application has a mysql/postgresql/oracle test settings, you can pass the settings file and run this to use your settings file:

```
DJANGO_SETTINGS_MODULE=<your test settings> python setup.py test
```

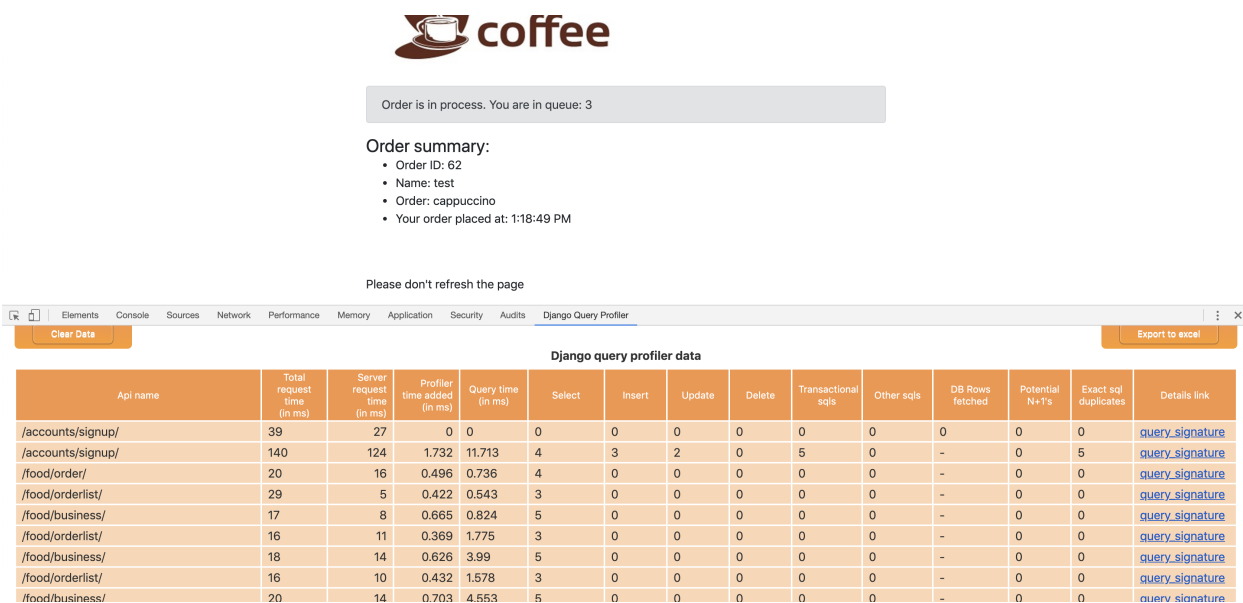
As an example, see tox file to see how we are passing this environment variable to run tests for mysql/postgres/sqlite

If you want to run tests against the full matrix of python version supported and django version supported, for all the drivers supported by Django, run:

```
tox
```

This is how we are also running tests on travisCI - so that all changes to the project are tested for all the above combinations

columns in chrome plugin



The screenshot shows a web application interface for a coffee shop. At the top, there is a logo for "coffee" with a coffee cup icon. Below the logo, a message states "Order is in process. You are in queue: 3". An "Order summary:" section lists the following details: Order ID: 62, Name: test, Order: cappuccino, and Your order placed at: 1:18:49 PM. A warning message "Please don't refresh the page" is displayed below the summary.

At the bottom of the screenshot, the Django Query Profiler is open, displaying a table of query performance data. The table has 15 columns: Api name, Total request time (in ms), Server request time (in ms), Profiler time added (in ms), Query time (in ms), Select, Insert, Update, Delete, Transactional sqls, Other sqls, DB Rows fetched, Potential N+1's, Exact sql duplicates, and Details link. The table contains 10 rows of data for various API endpoints.

Api name	Total request time (in ms)	Server request time (in ms)	Profiler time added (in ms)	Query time (in ms)	Select	Insert	Update	Delete	Transactional sqls	Other sqls	DB Rows fetched	Potential N+1's	Exact sql duplicates	Details link
/accounts/signup/	39	27	0	0	0	0	0	0	0	0	0	0	0	query signature
/accounts/signup/	140	124	1.732	11.713	4	3	2	0	5	0	-	0	5	query signature
/food/order/	20	16	0.496	0.736	4	0	0	0	0	0	-	0	0	query signature
/food/orderlist/	29	5	0.422	0.543	3	0	0	0	0	0	-	0	0	query signature
/food/business/	17	8	0.665	0.824	5	0	0	0	0	0	-	0	0	query signature
/food/orderlist/	16	11	0.369	1.775	3	0	0	0	0	0	-	0	0	query signature
/food/business/	18	14	0.626	3.99	5	0	0	0	0	0	-	0	0	query signature
/food/orderlist/	16	10	0.432	1.578	3	0	0	0	0	0	-	0	0	query signature
/food/business/	20	14	0.703	4.553	5	0	0	0	0	0	-	0	0	query signature

There are 15 columns in the chrome plugin table:

- Api name: This is the api name that we see in the network tab in chrome devtools
- Total request time (in ms): This is the total round-trip time of the request. This is also the same as what chrome network tab shows for that api
- Server request time (in ms): This is the time the request spends on the server - assuming that the *django-query-profiler* middleware is the first one in the list.
- Profiler time added (in ms): This is the overhead added by profiler to the request
- Query time (in ms): This is the total time taken by all queries for that request
- Select: Count of select sqls
- Insert: Count of insert sqls

- Update: Count of update sqls
- Delete: Count of delete sqls
- Transactional sqls: Count of begin/end transaction sqls
- Other sqls: Count of sqls that we were not able to classify as above five. Ideally this should never happen
- DB Rows fetched: Count of database rows that the select queries fetch from database. Note that sqlite does not return number of rows fetched, so it would show up as '-'
- Potential N+1's: This represents the count of N+1 queries that the profiler found. If this number seems high, the API is definitely something that should be optimized.
- Exact sql duplicates: This represents the count of queries which had the same (query, param) but was executed multiple times to the database. If this number is higher, consider doing *query caching*, or pulling the sql out of the loop
- Details link: This is the url that would show a detailed view, and the recommendation on how to fix the code path