django-sql-dashboard documentation

Release 1.2

Simon Willison

Dec 16, 2023

CONTENTS

CHAPTER

ONE

INSTALLATION AND CONFIGURATION

1.1 Install using pip

Install this library using `pip`:

```
$ pip install django-sql-dashboard
```

1.1.1 Run migrations

The migrations create tables that store dashboards and queries:

```
$ ./manage.py migrate
```

1.2 Configuration

Add "`django_sql_dashboard`" to your `INSTALLED_APPS` in `settings.py`.

Add the following to your `urls.py`:

```
from django.urls import path, include
import django_sql_dashboard

urlpatterns = [
    # ...
    path("dashboard/", include(django_sql_dashboard.urls)),
]
```

1.3 Setting up read-only PostgreSQL credentials

The safest way to use this tool is against a dedicated read-only replica of your database - see *security* for more details.

Create a new PostgreSQL user or role that is limited to read-only SELECT access to a specific list of tables.

If your read-only role is called `my-read-only-role`, you can grant access using the following SQL (executed as a privileged user):

```
GRANT USAGE ON SCHEMA PUBLIC TO "my-read-only-role";
```

This grants that role the ability to see what tables exist. You then need to grant `SELECT` access to specific tables like this:

```
GRANT SELECT ON TABLE
    public.locations_location,
    public.locations_county,
    public.django_content_type,
    public.django_migrations
TO "my-read-only-role";
```

Think carefully about which tables you expose to the dashboard - in particular, you should avoid exposing tables that contain sensitive data such as `auth_user` or `django_session`.

If you do want to expose `auth_user` - which can be useful if you want to join other tables against it to see details of the user that created another record - you can grant access to specific columns like so:

```
GRANT SELECT(
  id, last_login, is_superuser, username, first_name,
  last_name, email, is_staff, is_active, date_joined
) ON auth_user TO "my-read-only-role";
```

This will allow queries against everything except for the `password` column.

Note that if you use this pattern the query `select * from auth_user` will return a "permission denied" error. You will need to explicitly list the columns you would like to see from that table instead, for example `select id, username, date_joined from auth_user`.

1.4 Configuring the "dashboard" database alias

Django SQL Dashboard defaults to executing all queries using the `"dashboard"` Django database alias.

You can define this `"dashboard"` database alias in `settings.py`. Your `DATABASES` section should look something like this:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "mydb",
        "USER": "read_write_user",
        "PASSWORD": "read_write_password",
        "HOST": "dbhost.example.com",
        "PORT": "5432",
    },
    "dashboard": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "mydb",
        "USER": "read_only_user",
        "PASSWORD": "read_only_password",
        "HOST": "dbhost.example.com",
        "PORT": "5432",
        "OPTIONS": {
            "options": "-c default_transaction_read_only=on -c statement_timeout=100"
        },
```

(continues on next page)

(continued from previous page)

```
    },
}
```

In addition to the read-only user and password, pay attention to the `"OPTIONS"` section: this sets a statement timeout of 100ms - queries that take longer than that will be terminated with an error message. It also sets it so transactions will be read-only by default, as an extra layer of protection should your read-only user have more permissions that you intended.

Now visit `/dashboard/` as a staff user to start trying out the dashboard.

1.4.1 Danger mode: configuration without a read-only database user

Some hosting environments such as Heroku charge extra for the ability to create read-only database users. For smaller projects with dashboard access only made available to trusted users it's possible to configure this tool without a read-only account, using the following options:

```
    # ...
    "dashboard": {
        "ENGINE": "django.db.backends.postgresql",
        "USER": "read_write_user",
        # ...
        "OPTIONS": {
            "options": "-c default_transaction_read_only=on -c statement_timeout=100"
        },
    },
```

The `-c default_transaction_read_only=on` option here should prevent accidental writes from being executed, but note that dashboard users in this configuration will be able to access *all tables* including tables that might contain sensitive information. Only use this trick if you are confident you fully understand the implications!

1.4.2 dj-database-url and django-configurations

If you are using dj-database-url or django-configurations *(with `database` extra requirement)*, your `DATABASES` section should look something like this:

```
import dj_database_url

# ...

DATABASES = {
    "default": dj_database_url.config(env="DATABASE_URL"),
    "dashboard": dj_database_url.config(env="DATABASE_DASHBOARD_URL"),
}
```

You can define the two database url variables in your environment like this:

```
DATABASE_URL=postgresql://read_write_user:read_write_password@dbhost.example.com:5432/
↪mydb
DATABASE_DASHBOARD_URL=postgresql://read_write_user:read_write_password@dbhost.example.
↪com:5432/mydb?options=-c%20default_transaction_read_only%3Don%20-c%20statement_timeout
↪%3D100
```

1.5 Django permissions

Access to the `/dashboard/` interface is controlled by the Django permissions system. To grant a Django user or group access, grant them the `django_sql_dashboard.execute_sql` permission. This is displayed in the admin interface as:

```
django_sql_dashboard | dashboard | Can execute arbitrary SQL queries
```

Dashboard editing is currently handled by the Django admin interface. This means a user needs to have **staff** status (allowing them access to the Django admin interface) in order to edit one of their saved dashboards.

The regular Django permission for "can edit dashboard" is ignored. Instead, a permission system that is specific to Django SQL Dashboard is used to control edit permissions. See *Edit permissions* for details.

1.6 Additional settings

You can customize the following settings in Django's `settings.py` module:

- `DASHBOARD_DB_ALIAS = "db_alias"` - which database alias to use for executing these queries. Defaults to `"dashboard"`.
- `DASHBOARD_ROW_LIMIT = 1000` - the maximum number of rows that can be returned from a query. This defaults to 100.
- `DASHBOARD_UPGRADE_OLD_BASE64_LINKS` - prior to version 0.8a0 SQL URLs used base64-encoded JSON. If you set this to `True` any hits that include those old URLs will be automatically redirected to the upgraded new version. Use this if you have an existing installation of `django-sql-dashboard` that people already have saved bookmarks for.
- `DASHBOARD_ENABLE_FULL_EXPORT` - set this to `True` to enable the full results CSV/TSV export feature. It defaults to `False`. Enable this feature only if you are confident that the database alias you are using does not have write permissions to anything.
- `DASHBOARD_DISABLE_JSON` - set to `True` to disable the feature where `/dashboard/name-of-dashboard.json` provides a JSON representation of the dashboard. This defaults to `False`.

1.7 Custom templates

The templates used by `django-sql-dashboard` extend a base template called `django_sql_dashboard/base.html`, which provides Django template blocks named `title` and `content`. You can customize the appearance of your dashboard installation by providing your own version of this base template in your own configured `templates/` directory.

CHAPTER

TWO

RUNNING SQL QUERIES

Visit `/dashboard/` to get started. This interface allows you to execute one or more PostgreSQL SQL queries.

Results will be displayed below each query, limited to a maxmim of 100 rows.

The queries you have executed are encoded into the URL of the page. This means you can bookmark queries and share those links with other people who can access your dashboard.

Note that the queries in the URL are signed using Django's `SECRET_KEY` setting. This means that changing you secret will break your bookmarked URLs.

2.1 SQL parameters

If your SQL query contains `%(name)s` parameters, `django-sql-dashboard` will convert those into form fields on the page and allow users to submit values for them. These will be correctly quoted and escaped in the SQL query.

Given the following SQL query:

```
select * from blog_entry where slug = %(slug)s
```

A form field called `slug` will be displayed, and the user will be able to use that to search for blog entries with that given slug.

Here's a more advanced example:

```
select * from location
where state_id = cast(%(state_id)s as integer)
and name ilike '%%' || %(search)s || '%%';
```

Here a form will be displayed with `state_id` and `search` fields.

The values provided by the user will always be treated like strings - so in this example the `state_id` is cast to integer in order to be matched with an integer column.

Any `%` characters - for example in the `ilike` query above - need to be escaped by providing them twice: `%%`.

CHAPTER

THREE

SAVED DASHBOARDS

A set of SQL queries can be used to create a saved dashboard. Saved dashboards have URLs and support permissions, so you can specify which users are allowed to see which dashboard.

You can create a saved dashboard from the interactive dashboard interface (at `/dashboard/`) - execute some queries, then scroll down to the "Save this dashboard" form.

3.1 View permissions

The following viewing permission policies are available:

- `private`: Only the user who created (owns) the dashboard can view
- `public`: Any user can view
- `unlisted`: Any user can view, but they need to know the URL (this feature is not complete)
- `loggedin`: Any logged-in user can view
- `group`: Any user who is a member of the `view_group` attached to the dashboard can view
- `staff`: Any user who is staff can view
- `superuser`: Any user who is a superuser can view

3.2 Edit permissions

The edit policy controls which users are allowed to edit a dashboard - defaulting to the user who created that dashboard.

Editing currently takes place through the Django Admin - so only users who are staff members with access to that interface will be able to edit their dashboards.

The full list of edit policy options are:

- `private`: Only the user who created (owns) the dashboard can edit
- `loggedin`: Any logged-in user can edit
- `group`: Any user who is a member of the `edit_group` attached to the dashboard can edit
- `staff`: Any user who is staff can edit
- `superuser`: Any user who is a superuser can edit

Dashboards belong to the user who created them. Only Django super-users can re-assign ownership of dashboards to other users.

3.3 JSON export

If your dashboard is called `/dashboards/demo/` you can add `.json` to get `/dashboards/demo.json` which will return a JSON representation of the dashboard.

The JSON format looks something like this:


```
{
  "title": "Tag word cloud",
  "queries": [
    {
      "sql": "select \"tag\" as wordcloud_word, count(*) as wordcloud_count from (select␣
↪blog_tag.tag from blog_entry_tags join blog_tag on blog_entry_tags.tag_id = blog_tag.
↪id\r\nunion all\r\nselect blog_tag.tag from blog_blogmark_tags join blog_tag on blog_
↪blogmark_tags.tag_id = blog_tag.id\r\nunion all\r\nselect blog_tag.tag from blog_
↪quotation_tags join blog_tag on blog_quotation_tags.tag_id = blog_tag.id) as results␣
↪where tag != 'quora' group by \"tag\" order by wordcloud_count desc",
      "rows": [
        {
          "wordcloud_word": "python",
          "wordcloud_count": 826
        },
        {
          "wordcloud_word": "javascript",
          "wordcloud_count": 604
        },
        {
          "wordcloud_word": "django",
          "wordcloud_count": 529
        },
        {
          "wordcloud_word": "security",
          "wordcloud_count": 402
        },
        {
          "wordcloud_word": "datasette",
          "wordcloud_count": 331
        },
        {
          "wordcloud_word": "projects",
          "wordcloud_count": 282
        }
      ],
    }
  ]
}
```


Set the `DASHBOARD_DISABLE_JSON` setting to `True` to disable this feature.

CHAPTER

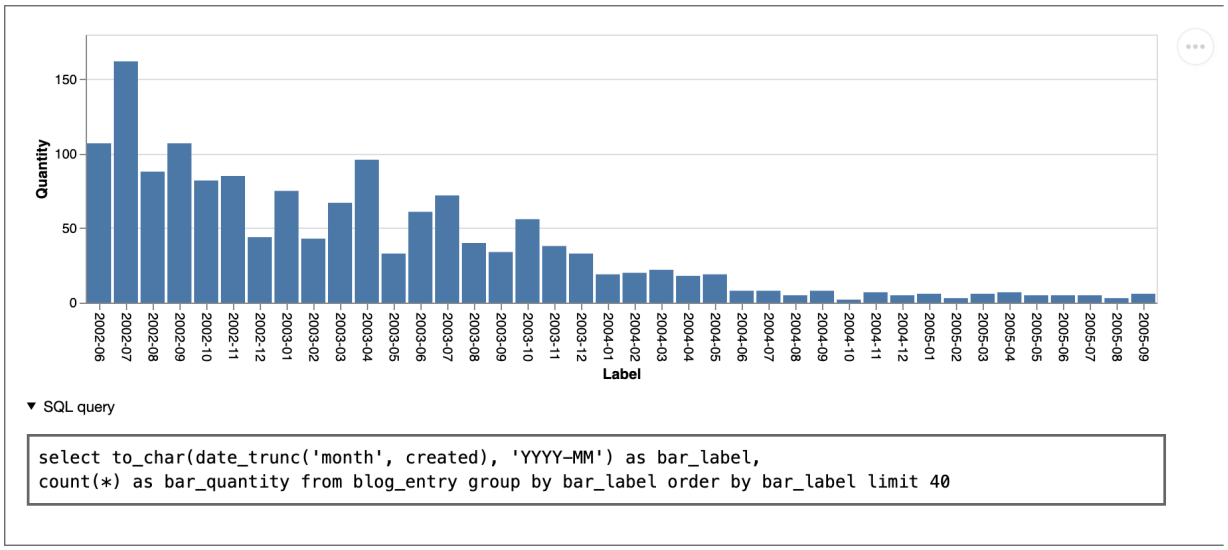
FOUR

WIDGETS

SQL queries default to displaying as a table. Other forms of display - called widgets - are also available, and are selected based on the names of the columns returned by the query.

4.1 Bar chart: bar_label, bar_quantity

A query that returns columns called `bar_label` and `bar_quantity` will be rendered as a simple bar chart, using Vega-Lite.



Bar chart live demo: simonwillison.net/dashboard/by-month/

SQL example:

```
select
  county.name as bar_label,
  count(*) as bar_quantity
from location
  join county on county.id = location.county_id
group by county.name
order by count(*) desc limit 10
```

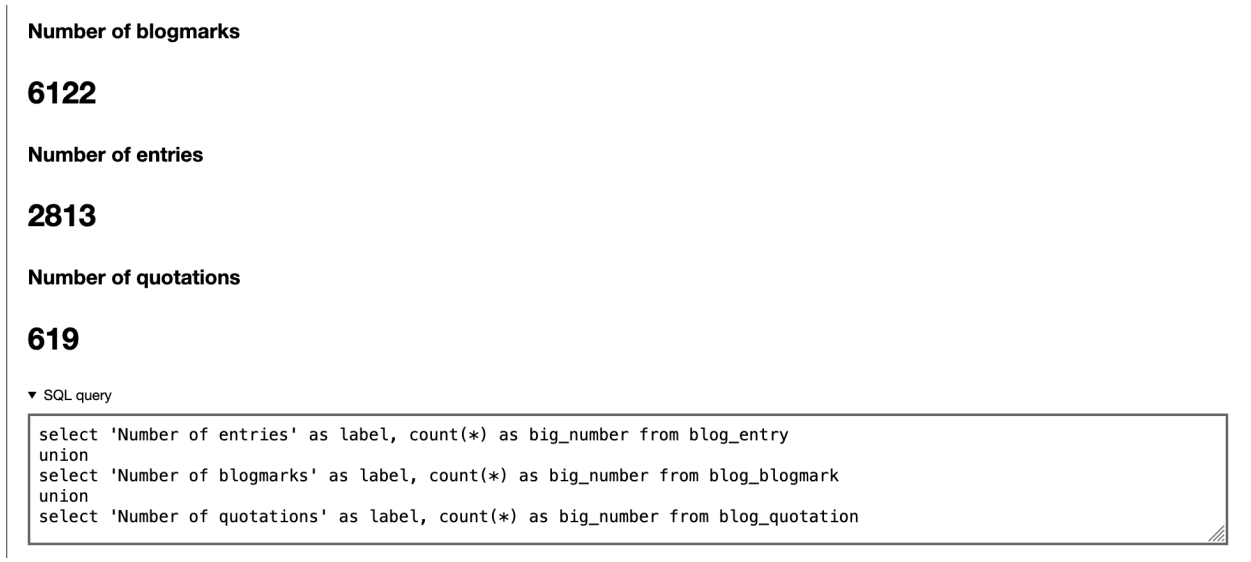
Or using a static list of values:

```
SELECT * FROM (
    VALUES (1, 'one'), (2, 'two'), (3, 'three')
) AS t (bar_quantity, bar_label);
```

4.2 Big number: big_number, label

If you want to display the results as a big number accompanied by a label, you can do so by returning `big_number` and `label` columns from your query, for example.

```
select 'Number of states' as label, count(*) as big_number from states;
```



Big number live demo: simonwillison.net/dashboard/big-numbers-demo/

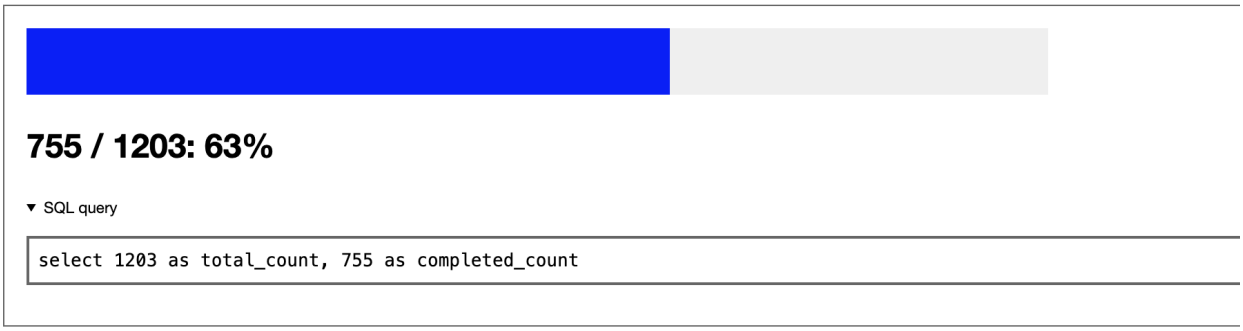
4.3 Progress bar: completed_count, total_count

To display a progress bar, return columns `total_count` and `completed_count`.

```
select 1203 as total_count, 755 as completed_count;
```

This SQL pattern can be useful for constructing progress bars:

```
select (
  select count(*) from task
) as total_count, (
  select count(*) from task where resolved_at is not null
) as completed_count
```



Progress bar live demo: simonwillison.net/dashboard/progress-bar-demo/

4.4 Word cloud: wordcloud_count, wordcloud_word

To display a word cloud, return a column `wordcloud_word` containing words with a corresponding `wordcloud_count` column with the frequency of those words.

This example generates word clouds for article body text:

```
with words as (
  select
    lower(
      (regexp_matches(body, '\w+', 'g'))[1]
    ) as word
  from
    articles
)
select
  word as wordcloud_word,
  count(*) as wordcloud_count
from
  words
group by
  word
order by
  count(*) desc
```

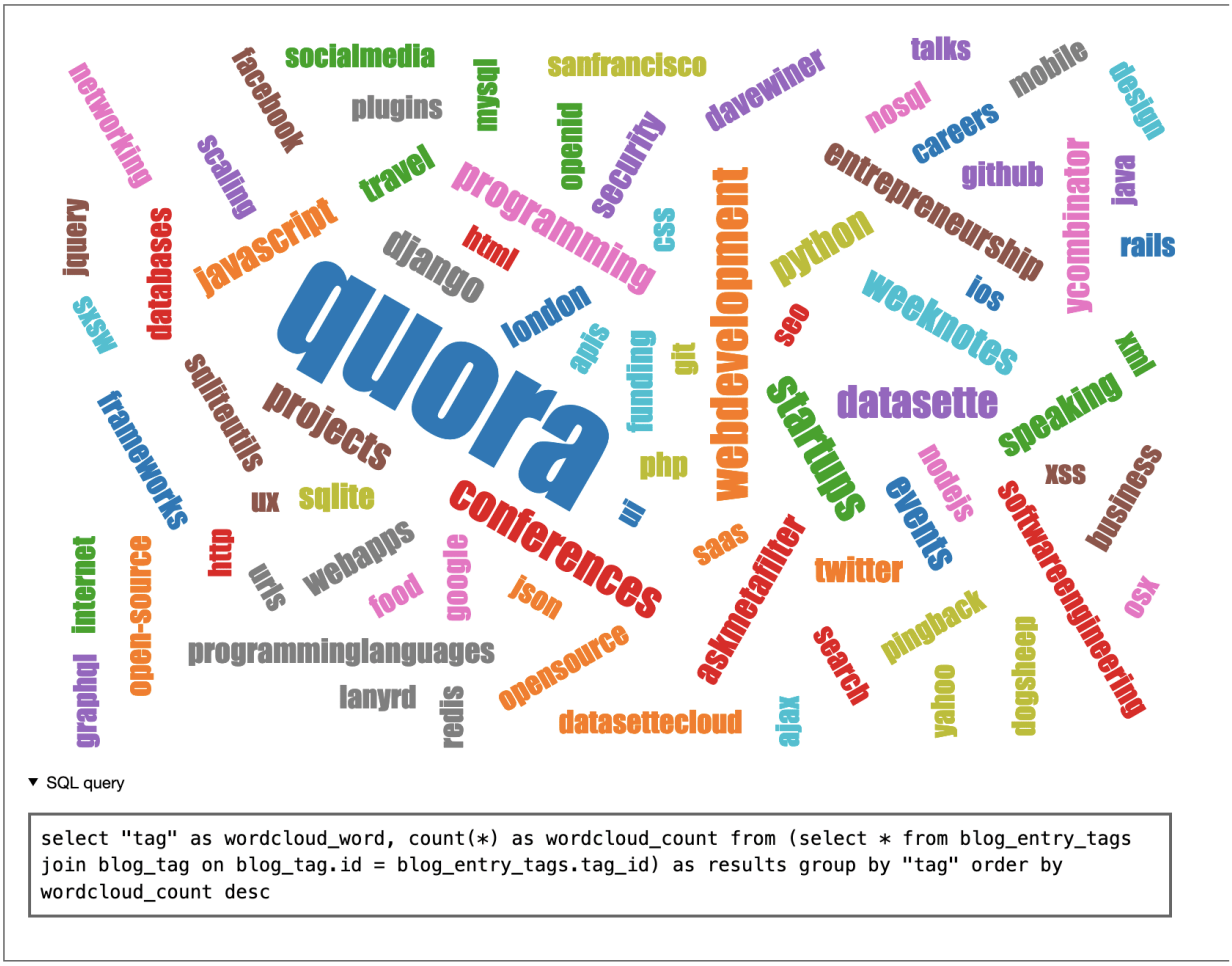
Here's a fun variant that uses PostgreSQL's built-in stemming algorithm to first remove common stop words:

```
with words as (
  select
    lower(
      (regexp_matches(to_tsvector('english', body)::text, '[a-z]+', 'g'))[1]
    ) as word
  from
    articles
)
select
  word as wordcloud_word,
  count(*) as wordcloud_count
from
```

(continues on next page)

(continued from previous page)

```
  words
group by
  word
order by
  count(*) desc
```



Word cloud live demo: simonwillison.net/dashboard/tag-cloud/

4.5 markdown

Return a single column called `markdown` to render the contents as Markdown, for example:

```
select '# Number of states: ' || count(*) as markdown from states;
```

4.6 html

Return a single column called `html` to render the contents directly as HTML. This HTML is filtered using Bleach so the only tags allowed are `a[href]`, `abbr`, `acronym`, `b`, `blockquote`, `code`, `em`, `i`, `li`, `ol`, `strong`, `ul`, `pre`, `p`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`.

```
select '<h1>Number of states: ' || count(*) || '</h1>' as html from states;
```

CHAPTER

FIVE

CUSTOM WIDGETS

You can define your own custom widgets by creating templates with special names.

Decide on the column names that you wish to customize for, then sort them alphabetically and join them with hyphens to create your template name.

For example, you could define a widget that handles results returned as `placename`, `geojson` by creating a template called `geojson-placename.html`.

Save that in one of your template directories as `django_sql_dashboard/widgets/geojson-placename.html`.

Any SQL query that returns exactly the columns `placename` and `geojson` will now be rendered by your custom template file.

Within your custom template you will have access to a template variable called `result` with the following keys:

- `result.sql` - the SQL query that is being displayed
- `rows` - a list of rows, where each row is a dictionary mapping columns to their values
- `row_lists` - a list of rows, where each row is a list of the values in that row
- `description` - the psycopg2 cursor description
- `columns` - a list of string column names
- `column_details` - a list of `{"name": column_name, "is_unambiguous": True or False}` dictionaries - `is_unambiguous` is `False` if multiple columns of the same name are returned by this query
- `truncated` - boolean, specifying whether the results were truncated (at 100 items) or not
- `extra_qs` - extra parameters for the page encoded as a query string fragment - so if the page was loaded with `state_id=5` then `extra_qs` would be `&state_id=5`. You can use this to assemble links to further queries, like the "Count" column links in the default table view.
- `duration_ms` - how long the query took, in floating point milliseconds
- `templates` - a list of templates that were considered for rendering this widget

The easiest way to define your custom widget template is to extend the `django_sql_dashboard/widgets/_base_widget.html` base template.

Here is the full implementation of the `big_number`, `label` widget that is included with Django SQL Dashboard, in the `django_sql_dashboard/widgets/big_number-label.html` template file:

```
{% extends "django_sql_dashboard/widgets/_base_widget.html" %}

{% block widget_results %}
  {% for row in result.rows %}
    <div class="big-number">
```

(continues on next page)

(continued from previous page)

```
      <p><strong>{{ row.label }}</strong></p>
      <h1>{{ row.big_number }}</h1>
    </div>
  {% endfor %}
{% endblock %}
```

You can find more examples of widget templates in the templates/django_sql_dashboard/widgets directory.

CHAPTER

SIX

SECURITY

Allowing people to execute their own SQL directly against your database is risky business!

The safest way to use this tool is to create a read-only replica of your PostgreSQL database with a read-only role that enforces a statement time-limit for executed queries. Different database providers have different mechanisms for doing this - consult your hosting provider's documentation.

You should only provide access to this tool to people you trust. Malicious users may be able to negatively affect the performance of your servers through constructing SQL queries that deliberately consume large amounts of resources.

Configured correctly, Django SQL Dashboard uses a number of measures to keep your data and your database server safe:

- I strongly recommend creating a dedicated PostgreSQL role for accessing your database with read-only permissions granted to an allow-list of tables. PostgreSQL has extremely robust, well tested permissions which this tool can take full advantage of.
- Likewise, configuring a PostgreSQL-enforced query time limit can reduce the risk of expensive queries affecting the performance of the rest of your site.
- Setting up a read-only reporting replica for use with this tool can provide even stronger isolation from other site traffic.
- Your allow-list of tables should not include tables with sensitive information. Django's auth_user table contains password hashes, and the django_session table contains user session information. Neither should be exposed using this tool.
- Access to the dashboard is controlled by Django's permissions system, which means you can limit access to trusted team members.
- SQL queries can be passed to the dashboard using a ?sql= query string parameter - but this parameter needs to be signed before it will be executed. This should prevent attempts to trick you into executing malevolent SQL queries by sending you crafted links - while still allowing your team to create links to queries that can be securely shared.
- Any time a user views a dashboard page while logged in, `Cache-Control: private` is set on the response to ensure the authenticated dashboard will not be stored in any intermediary HTTP caches

CHAPTER

SEVEN

CONTRIBUTING

To contribute to this library, first checkout the code. Then create a new virtual environment:

```
cd django-sql-dashboard
python -m venv venv
source venv/bin/activate
```

Or if you are using `pipenv`:

```
pipenv shell
```

Now install the dependencies and tests:

```
pip install -e '.[test]'
```

7.1 Running the tests

To run the tests:

```
pytest
```

7.2 Generating new migrations

To generate migrations for model changes:

```
cd test_project
./manage.py makemigrations
```

7.3 Code style

This library uses Black for code formatting. The correct version of Black will be installed by `pip install -e '.[test]'` - you can run `black .` in the root directory to apply those formatting rules.

7.4 Documentation

Documentation for this project uses MyST - it is written in Markdown and rendered using Sphinx.

To build the documentation locally, run the following:

```
cd docs
pip install -r requirements.txt
make livehtml
```

This will start a live preview server, using sphinx-autobuild.

7.5 Using Docker Compose

If you're familiar with Docker–or even if you're not–you may want to consider using our optional Docker Compose setup.

An advantage of this approach is that it relieves you of setting up any dependencies, such as ensuring that you have the proper version of Python and Postgres and so forth. On the downside, however, it does require you to familiarize yourself with Docker, which, while relatively easy to use, still has its own learning curve.

To try out the Docker Compose setup, you will first want to get Docker and install Docker Compose.

Then, after checking out the code, run the following:

```
cd django-sql-dashboard
docker-compose build
```

At this point, you can start editing code. To run any development tools such as `pytest` or `black`, just prefix everything with `docker-compose run app`. For instance, to run the test suite, run:

```
docker-compose run app python pytest
```

If this is a hassle, you can instead run a bash shell inside your container:

```
docker-compose run app bash
```

At this point, you'll be in a bash shell inside your container, and can run development tools directly.

7.5.1 Using the dashboard interactively

The Docker Compose setup is configured to run a simple test project that you can use to tinker with the dashboard interactively.

To use it, run:

```
docker-compose up
```

Then, in a separate terminal, run:

```
docker-compose run app python test_project/manage.py createsuperuser
```

You will now be prompted to enter details about a new superuser. Once you've done that, you can visit the example app's dashboard at http://localhost:8000/. After entering the credentials for the superuser you just created, you will be able to tinker with the dashboard.

7.5.2 Editing the documentation

Running `docker-compose up` also starts the documentation system's live preview server. You can visit it at http://localhost:8001/.

7.5.3 Changing the default ports

If you are already using ports 8000 and/or 8001 for other things, you can change them. To do this, create a file in the repository root called `.env` and populate it with the following:

```
APP_PORT=9000
DOCS_PORT=9001
```

You can change the above port values to whatever makes sense for your setup.

Once you next run `docker-compose up` again, the services will be running on the ports you specified in `.env`.

7.5.4 Changing the default UID and GID

The default settings assume that the user id (UID) and group id (GID) of the account you're using to develop are both 1000. This is likely to be the case, since that's the UID/GID of the first non-root account on most systems. However, if your account doesn't match this, you can customize the container to use a different UID/GID.

For instance, if your UID and GID are 1001, you can build your container with the following arguments:

```
docker-compose build --build-arg UID=1001 --build-arg GID=1001
```

7.5.5 Updating

The project's Python dependencies are all baked into the container image, which means that whenever they change (or to be safe, whenever you `git pull` new changes to the codebase), you will want to run:

```
docker-compose build
```

You will also want to restart `docker-compose up`.

7.5.6 Cleaning up

If you somehow get your Docker Compose setup into a broken state, or you decide that you never use Docker Compose again, you can clean everything up by running:

```
docker-compose down -v
```

CHAPTER

EIGHT

DJANGO-SQL-DASHBOARD

Django SQL Dashboard provides an authenticated interface for executing read-only SQL queries directly against your PostgreSQL database, bringing a useful subset of Datasette to Django.

Applications include ad-hoc analysis and debugging, plus the creation of reporting dashboards that can be shared with team members or published online.

See my blog for more about this project, including a video demo.

Features include:

- Safely run read-only one or more SQL queries against your database and view the results in your browser
- Bookmark queries and share those links with other members of your team
- Create saved dashboards from your queries, with full control over who can view and edit them
- Named parameters such as `select * from entries where id = %(id)s` will be turned into form fields, allowing quick creation of interactive dashboards
- Produce bar charts, progress bars and more from SQL queries, with the ability to easily create new custom dashboard widgets using the Django template system
- Write SQL queries that safely construct and render markdown and HTML
- Export the full results of a SQL query as a downloadable CSV or TSV file, using a combination of Django's streaming HTTP response mechanism and PostgreSQL server-side cursors to efficiently stream large amounts of data without running out of resources
- Copy and paste the results of SQL queries directly into tools such as Google Sheets or Excel
- Uses Django's authentication system, so dashboard accounts can be granted using Django's Admin tools

8.1 Documentation

Full documentation is at django-sql-dashboard.datasette.io

8.2 Screenshot

8.3 Alternatives

- django-sql-explorer provides a related set of functionality that also works against database backends other than PostgreSQL