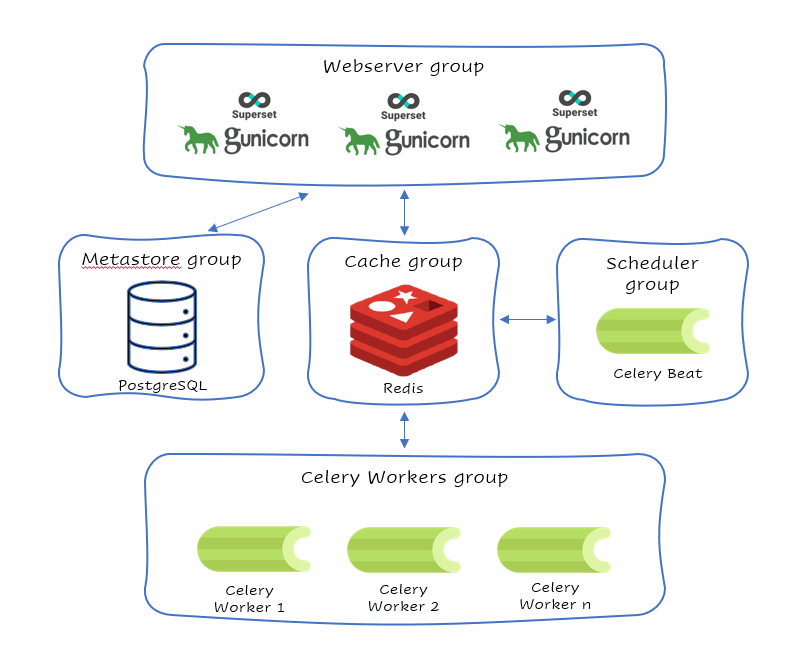
# Superset

## Architecture



A Superset installation is made up of these components:

1. The Superset application itself
2. A metadata database
3. A caching layer (optional, but necessary for some features)
4. A worker & beat (optional, but necessary for some features)

### Optional components and associated features

The optional components above are necessary to enable these features:

* [Alerts and Reports](https://superset.apache.org/docs/configuration/alerts-reports)
* [Caching](https://superset.apache.org/docs/configuration/cache)
* [Async Queries](https://superset.apache.org/docs/configuration/async-queries-celery/)
* [Dashboard Thumbnails](https://superset.apache.org/docs/configuration/cache/#caching-thumbnails)

### The Superset Application

* A user visits a chart or dashboard
* That triggers a SQL query to the data warehouse holding the underlying dataset
* The resulting data is served up in a data visualization
* The Superset application is comprised of the Python (Flask) backend application (server), API layer, and the React frontend, built via Webpack, and static assets needed for the application to work

### Metadata Database

This is where chart and dashboard definitions, user information, logs, etc. are stored. Superset is tested to work with PostgreSQL and MySQL databases as the metadata database (not be confused with a data source like your data warehouse, which could be a much greater variety of options like Snowflake, Redshift, etc.).

Some installation methods like our Quickstart and PyPI come configured by default to use a SQLite on-disk database. And in a Docker Compose installation, the data would be stored in a PostgreSQL container volume. Neither of these cases are recommended for production instances of Superset.

### Caching Layer

The caching layer serves two main functions:

* Store the results of queries to your data warehouse so that when a chart is loaded twice, it pulls from the cache the second time, speeding up the application and reducing load on your data warehouse.
* Act as a message broker for the worker, enabling the Alerts & Reports, async queries, and thumbnail caching features.

Most people use Redis for their cache, but Superset supports other options too. See the [cache docs](https://superset.apache.org/docs/configuration/cache/) for more.

The following cache configurations can be customized in this way:

* Dashboard filter state (required): FILTER\_STATE\_CACHE\_CONFIG.
* Explore chart form data (required): EXPLORE\_FORM\_DATA\_CACHE\_CONFIG
* Metadata cache (optional): CACHE\_CONFIG
* Charting data queried from datasets (optional): DATA\_CACHE\_CONFIG

### Worker and Beat

This is one or more workers who execute tasks like run async queries or take snapshots of reports and send emails, and a "beat" that acts as the scheduler and tells workers when to perform their tasks. Most installations use Celery for these components.

#### Async Queries via Celery

On large analytic databases, it’s common to run queries that execute for minutes or hours. To enable support for long running queries that execute beyond the typical web request’s timeout (30-60 seconds), it is necessary to configure an asynchronous backend for Superset which consists of:

* one or many Superset workers (which is implemented as a Celery worker), and can be started with the celery worker command, run celery worker --help to view the related options.
* a celery broker (message queue) for which we recommend using Redis or RabbitMQ
* a results backend that defines where the worker will persist the query results

class CeleryConfig(object):  
 broker\_url = "redis://localhost:6379/0"  
 imports = (  
 "superset.sql\_lab",  
 "superset.tasks.scheduler",  
 )  
 result\_backend = "redis://localhost:6379/0"  
 worker\_prefetch\_multiplier = 10  
 task\_acks\_late = True  
 task\_annotations = {  
 "sql\_lab.get\_sql\_results": {  
 "rate\_limit": "100/s",  
 },  
 }  
  
CELERY\_CONFIG = CeleryConfig

## Celery Flower

Flower is a web based tool for monitoring the Celery cluster which you can install from pip:

pip install flower

You can run flower using:

celery --app=superset.tasks.celery\_app:app flower

## StatsD Logging

Superset can be configured to log events to [StatsD](https://github.com/statsd/statsd) if desired. Most endpoints hit are logged as well as key events like query start and end in SQL Lab.

To setup StatsD logging, it’s a matter of configuring the logger in your superset\_config.py.

from superset.stats\_logger import StatsdStatsLogger  
STATS\_LOGGER = StatsdStatsLogger(host='localhost', port=8125, prefix='superset')

### Adding an initial SECRET\_KEY

Superset requires a user-specified SECRET\_KEY to start up. You can generate a strong secure key with openssl rand -base64 42.

This key will be used for securely signing session cookies and encrypting sensitive information stored in Superset's application metadata database. Your deployment must use a complex, unique key.

SECRET\_KEY = 'YOUR\_OWN\_RANDOM\_GENERATED\_SECRET\_KEY'

### Setting up a production metadata database

By default, Superset is configured to use [SQLite](https://www.sqlite.org/), a self-contained, single-file database that offers a simple and fast way to get started.

SQLALCHEMY\_DATABASE\_URI

postgresql://<UserName>:<DBPassword>@<Database Host>/<Database Name>

### Running on a WSGI HTTP Server

While you can run Superset on NGINX or Apache, we recommend using Gunicorn in async mode. This enables impressive concurrency even and is fairly easy to install and configure.

-w 10 \  
 -k gevent \  
 --worker-connections 1000 \  
 --timeout 120 \  
 -b 0.0.0.0:6666 \  
 --limit-request-line 0 \  
 --limit-request-field\_size 0 \  
 --statsd-host localhost:8125 \  
 "superset.app:create\_app()"

### HTTPS Configuration

You can configure HTTPS upstream via a load balancer or a reverse proxy (such as nginx) and do SSL/TLS Offloading before traffic reaches the Superset application. In this setup, local traffic from a Celery worker taking a snapshot of a chart for Alerts & Reports can access Superset at a http:// URL, from behind the ingress point. You can also configure [SSL in Gunicorn](https://docs.gunicorn.org/en/stable/settings.html#ssl) (the Python webserver) if you are using an official Superset Docker image.

### Configuration Behind a Load Balancer

If you are running superset behind a load balancer or reverse proxy (e.g. NGINX or ELB on AWS), you may need to utilize a healthcheck endpoint so that your load balancer knows if your superset instance is running. This is provided at /health which will return a 200 response containing “OK” if the webserver is running.

If the load balancer is inserting X-Forwarded-For/X-Forwarded-Proto headers, you should set ENABLE\_PROXY\_FIX = True in the superset config file (superset\_config.py) to extract and use the headers.

In case the reverse proxy is used for providing SSL encryption, an explicit definition of the X-Forwarded-Proto may be required. For the Apache webserver this can be set as follows: