We use cookies to make interactions with our websites and services easy and meaningful, to better understand how they are used and to tailor advertising. You can read more (https://www.salesforce.com/company/privacy/full_privacy.jsp#nav_info) and make your cookie choices here (https://www.salesforce.com/company/privacy/full_privacy.jsp#nav_info). By continuing to use this site you are giving us your consent to do this.

×

Deploying Python Applications with Gunicorn

Last updated 18 September 2018

Table of Contents

Web applications that process incoming HTTP requests concurrently make much more efficient use of dyno resources than web applications that only process one request at a time. Because of this, we recommend using web servers that support concurrent request processing whenever developing and running production services.

The Django and Flask web frameworks feature convenient built-in web servers, but these blocking servers only process a single request at a time. If you deploy with one of these servers on Heroku, your dyno resources will be underutilized and your application will feel unresponsive.

Gunicorn (http://gunicorn.org) is a pure-Python HTTP server for WSGI applications. It allows you to run any Python application concurrently by running multiple Python processes within a single dyno. It provides a perfect balance of performance, flexibility, and configuration simplicity.

This guide will walk you through deploying a new Python application to Heroku using the Gunicorn web server. For basic setup and knowledge about Heroku, see Getting Started with Python (https://devcenter.heroku.com/articles/getting-started-with-python).

⚠ As always, test configuration changes in a staging environment before you deploy to your production application.

Adding Gunicorn to your application

First, install Gunicorn with `pip`:

```
$ pip install gunicorn
```

Be sure to add `gunicorn` to your `requirements.txt` file as well.

Next, revise your application's `Procfile` to use Gunicorn. Here's an example `Procfile` for the Django application we created in Getting Started with Python on Heroku (https://devcenter.heroku.com/articles/getting-started-with-python).

Procfile

```
web: gunicorn gettingstarted.wsgi
```

Basic configuration

Gunicorn forks multiple system processes within each dyno to allow a Python app to support multiple concurrent requests without requiring them to be thread-safe. In Gunicorn terminology, these are referred to as worker processes (not to be confused with Heroku worker processes, which run in their own dynos).

Each forked system process consumes additional memory. This limits how many processes you can run in a single dyno. With a typical Django application memory footprint, you can expect to run 2–4 Gunicorn worker processes on a `free` , `hobby` or `standard-1x` dyno. Your application may allow for a variation of this, depending on your application's specific memory requirements.

We recommend setting a configuration variable for this setting. Gunicorn automatically honors the `WEB_CONCURRENCY` environment variable, if set.

```
$ heroku config:set WEB_CONCURRENCY=3
```

The `WEB_CONCURRENCY` environment variable is automatically set by Heroku, based on the processes' Dyno size. This feature is intended to be a sane starting point for your application. We recommend knowing the memory requirements of your processes and setting this configuration variable accordingly.

Procfile

```
web: gunicorn hello:app
```

The Heroku Labs log-runtime-metrics (https://devcenter.heroku.com/articles/log-runtime-metrics) feature adds support for enabling visibility into load and memory usage for running dynos. Once enabled, your can monitor application memory usage with the `heroku logs` command.

Advanced configuration

App preloading

If you are constrained for memory or experiencing slow app boot time, you might want to consider enabling the `preload` option. This loads the application code before the worker processes are forked.

```
web: gunicorn hello:app ––preload
```

See the Gunicorn Docs on Preloading (http://docs.gunicorn.org/en/latest/settings.html#preload-app) for more information.

Worker timeouts

By default, Gunicorn gracefully restarts a worker if hasn't completed any work within the last 30 seconds. If you expect your application to respond quickly to constant incoming flow of requests, try experimenting with a lower timeout configuration.

```
$ gunicorn hello:app --timeout 10
```

See the Gunicorn Docs on Worker Timeouts (http://docs.gunicorn.org/en/latest/settings.html#timeout) for more information.

Max request recycling

If your application suffers from memory leaks, you can configure Gunicorn to gracefully restart a worker after it has processed a given number of requests. This can be a convenient way to help limit the effects of the memory leak.

```
$ gunicorn hello:app --max-requests 1200
```

See the Gunicorn Docs on Max Requests (http://docs.gunicorn.org/en/latest/settings.html#max-requests) for more information.

Further reading

- Gunicorn Documentation (http://docs.gunicorn.org/en/latest/settings.html)