Logging, Flask, and Gunicorn... the Manageable Way

Posted 5 years ago by **Thomas Stringer** • Updated 3 years ago

Logging is one topic that some (many?) find boring. But something we can all agree on is that it is absolutely vital to software development and operations. Beginners to Flask (a lightweight but powerful Python web framework) may be disappointed to find that `print()` doesn't do exactly what they'd hope it would do, like in their CLI applications.

Flask requires that we rely heavily on the native logging functionality of Python. But when we stack a different WSGI (web server gateway interface) HTTP server on top of Flask, the confusion gets even more… confusing.

Native Flask logging

Forget about Gunicorn (a great, production-quality WSGI HTTP server) for a minute. Let's take a very simple Flask application all by itself:

```python
import logging
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')
def default_route():
    """Default route"""
    app.logger.debug('this is a DEBUG message')
    app.logger.info('this is an INFO message')
    app.logger.warning('this is a WARNING message')
    app.logger.error('this is an ERROR message')
    app.logger.critical('this is a CRITICAL message')
    return jsonify('hello world')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000, debug=True)
```

Running this application with `python app.py` (provided you named the above module the same), and calling `curl localhost:8000` from another process, the output of this Flask application should look similar to the following:

```
(venv) ~/dev/python/flasklogging
 $ python app.py
 * Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 308-334-981
---------------------------------------------------------------------------
DEBUG in app [app.py:17]:
this is a DEBUG message
---------------------------------------------------------------------------
---------------------------------------------------------------------------
INFO in app [app.py:18]:
this is an INFO message
---------------------------------------------------------------------------
---------------------------------------------------------------------------
WARNING in app [app.py:19]:
this is a WARNING message
---------------------------------------------------------------------------
---------------------------------------------------------------------------
ERROR in app [app.py:20]:
this is an ERROR message
---------------------------------------------------------------------------
---------------------------------------------------------------------------
CRITICAL in app [app.py:21]:
this is a CRITICAL message
---------------------------------------------------------------------------
127.0.0.1 - - [20/Jan/2018 17:38:31] "GET / HTTP/1.1" 200 -
```

What we're seeing above is Werkzeug (a WSGI utility library for Python, which Flask uses out-of-the-box) output.

Enter Gunicorn

Although Flask's built-in WSGI is sufficient for development, it's definitely not going to cut it in production. This is where Gunicorn comes into the picture. The important part here, though, is that Gunicorn has its own loggers and handlers. We need to wire up our Flask application to use those handlers so that all of our output, web application and WSGI, goes to the same place:

```
1  gunicorn_logger = logging.getLogger('gunicorn.error')
2  app.logger.handlers = gunicorn_logger.handlers
```

But what happens when we run this exact same code through Gunicorn and curl again?

```
1  $ gunicorn --workers=2 --bind=0.0.0.0:8000 app:app
```

```
^C(venv) ~/dev/python/flasklogging
 $ gunicorn --workers=2 --bind=0.0.0.0:8000 app:app
[2018-01-20 17:58:32 -0500] [18844] [INFO] Starting gunicorn 19.7.1
[2018-01-20 17:58:32 -0500] [18844] [INFO] Listening at: http://0.0.0.0:8000 (18
844)
[2018-01-20 17:58:32 -0500] [18844] [INFO] Using worker: sync
[2018-01-20 17:58:32 -0500] [18847] [INFO] Booting worker with pid: 18847
[2018-01-20 17:58:32 -0500] [18849] [INFO] Booting worker with pid: 18849
[2018-01-20 17:58:52,366] ERROR in app: this is an ERROR message
[2018-01-20 17:58:52,367] CRITICAL in app: this is a CRITICAL message
```

Hmmm… looks like only our *error* and *critical* log messages came through, but not *debug*, *info*, and *warning* messages.

There are a couple of reasons behind this: Gunicorn has its own loggers, and it's controlling log level through that mechanism. A fix for this would be to add `app.logger.setLevel(logging.DEBUG)`. But what's the problem with this approach? Well, first off, that's hard-coded into the application itself. Yes, we could refactor that out into an environment variable, but then we have **two different log levels**: one for the Flask application, but a totally separate one for Gunicorn, which is set through the –log-level parameter (values like "debug", "info", "warning", "error", and "critical").

The solution

What I've found to be a great solution to solve this problem is the following snippet (meant for your Flask application):

```
1  if __name__ != '__main__':
2      gunicorn_logger = logging.getLogger('gunicorn.error')
3      app.logger.handlers = gunicorn_logger.handlers
4      app.logger.setLevel(gunicorn_logger.level)
```

There are a few things at play here. By testing if **name** is equal to "**main**", that's a good wayto see if it's being run directly, or not. And the "not" would mean running my Python Flask application through Gunicorn in my workflow.

Then the next line (line #2 in the above snippet) we get a Logger object to the `gunicorn.error` logger. The key thing here (line #3) is to set the handlers of our Flask application logger to the Gunicorn logger (using the same output handlers and giving us a consistent logging experience).

The last line of that snippet is significant. When you pass –log-level to Gunicorn, that is going to (unsurprisingly) be the log level for its appropriate handler. By letting that trickle down to the Flask application logger's logging level, we now have a single source of truth for log levels: The Gunicorn logging level.

Now when we set `--log-level=warning` when invoking Gunicorn, this same logging level is used for Flask's logger. The full sample code of this example is as follows:

```python
import logging
from flask import Flask, jsonify

app = Flask(__name__)

if __name__ != '__main__':
    gunicorn_logger = logging.getLogger('gunicorn.error')
    app.logger.handlers = gunicorn_logger.handlers
    app.logger.setLevel(gunicorn_logger.level)

@app.route('/')
def default_route():
    """Default route"""
    app.logger.debug('this is a DEBUG message')
    app.logger.info('this is an INFO message')
    app.logger.warning('this is a WARNING message')
    app.logger.error('this is an ERROR message')
    app.logger.critical('this is a CRITICAL message')
    return jsonify('hello world')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000, debug=True)
```

Now when we run `gunicorn --workers=2 --bind=0.0.0.0:8000 --log-level=debug app:app` we not only get the Gunicorn debug logs, but the same logging level for our Flask application:

```
[2018-01-20 18:55:44 -0500] [21378] [INFO] Starting gunicorn 19.7.1
[2018-01-20 18:55:44 -0500] [21378] [DEBUG] Arbiter booted
[2018-01-20 18:55:44 -0500] [21378] [INFO] Listening at: http://0.0.0.0:8000 (21
378)
[2018-01-20 18:55:44 -0500] [21378] [INFO] Using worker: sync
[2018-01-20 18:55:44 -0500] [21381] [INFO] Booting worker with pid: 21381
[2018-01-20 18:55:44 -0500] [21382] [INFO] Booting worker with pid: 21382
[2018-01-20 18:55:44 -0500] [21378] [DEBUG] 2 workers
[2018-01-20 18:55:55 -0500] [21381] [DEBUG] GET /
[2018-01-20 18:55:55 -0500] [21381] [DEBUG] this is a DEBUG message
[2018-01-20 18:55:55 -0500] [21381] [INFO] this is an INFO message
[2018-01-20 18:55:55 -0500] [21381] [WARNING] this is a WARNING message
[2018-01-20 18:55:55 -0500] [21381] [ERROR] this is an ERROR message
[2018-01-20 18:55:55 -0500] [21381] [CRITICAL] this is a CRITICAL message
```

And if we specify a higher logging level, such as “warning”, we only get the warning (and above) logging messages from both Gunicorn *and* our Flask application:

```
$ gunicorn --workers=0 --bind=0.0.0.0:8000 --log-level=warning app:app
```

```
(venv) ~/dev/python/flasklogging
 $ gunicorn --workers=2 --bind=0.0.0.0:8000 --log-level=warning app:app
[2018-01-20 18:57:48 -0500] [22584] [WARNING] this is a WARNING message
[2018-01-20 18:57:48 -0500] [22584] [ERROR] this is an ERROR message
[2018-01-20 18:57:48 -0500] [22584] [CRITICAL] this is a CRITICAL message
```

Summary

The solution is simple but effective: Check to see if our Flask application is being run directly or through Gunicorn, and then set your Flask application logger’s handlers to the same as Gunicorn’s. And then finally, have a single logging level between Gunicorn and the Flask application.

Flask logging made easy! Enjoy!