python-dockerflow Documentation

Release 2017.4.0.post9

Mozilla Foundation

May 31, 2017

Contents

This package implements a few helpers and tools for Mozilla's Dockerflow pattern. The documentation can be found on python-dockerflow.readthedocs.io

CHAPTER 1

Installation

Please install the package using your favorite package installer:

```
pip install dockerflow
```

CHAPTER 2

Issues & questions

See the issue tracker on GitHub to open tickets if you have issues or questions about python-dockerflow.

CHAPTER 3

Dockerflow?

You may be asking 'What is Dockerflow?'

Here's what it's documentation says:

> Dockerflow is a specification for automated building, testing and publishing of docker web application images that comply to a common set of behaviours. Compliant images are simpler to deploy, monitor and manage in production.

CHAPTER 4

Features

environment Accept its configuration through environment variables. See: *Django*

port Listen on environment variable `$PORT` for HTTP requests. See: *Django*

version Must have a JSON version object at `/app/version.json`. See: *Django*

health

- Respond to `/__version__` with the contents of /app/version.json
- Respond to `/__heartbeat__` with a HTTP 200 or 5xx on error. This should check backing services like a database for connectivity
- Respond to `/__lbheartbeat__` with an HTTP 200. This is for load balancer checks and should not check backing services.

See: *Django*

logging Send text logs to `stdout` or `stderr`.

See: *Logging* for logging

See: *Django* for logging with Django

static content Serve its own static content. See: *Django*.

CHAPTER 5

Contents

Authors

- Peter Bengtsson (@peterbe)
- Mike Cooper (@mythmon)
- Will Kahn-Greene (@willkg)
- Michael Kelly (@Osmose)
- Jannis Leidel (@jezdez)
- Les Orchard (@lmorchard)

Changelog

2017.5.0 (2017-05-31)

- Improve logging documentation, thanks @willkg.
- Speed up timestamp calculation, thanks @peterbe.
- Make user id calculation compatible with Django >= 1.10.

2017.4.0 (2017-04-09)

- Ensure log formatter doesn't fail with non json-serializable parameters. Thanks @diox!

2017.1.1 (2017-01-25)

- Fixed PyPI deploy via Travis (added whl files).

2017.1.0 (2017-01-25)

- Replaced custom URL patterns in the Django support with new DockerflowMiddleware that also takes care of the "request.summary" logging.
- Added Python 3.6 to test harness.
- Fixed Flake8 tests.

2016.11.0 (2016-11-18)

- Added initial implementation for Django health checks based on Normandy and ATMO code. Many thanks to Mike Cooper for inspiration and majority of implementation.
- Added logging formatter and request.summary populating middleware, from the mozilla-cloud-services-logger project that was originally written by Les Orchard. Many thanks for the permission to re-use that code.
- Added documentation:

  https://python-dockerflow.readthedocs.io/

- Added Travis continous testing:

  https://travis-ci.org/mozilla-serviers/python-dockerflow

Logging

python-dockerflow provides a *dockerflow.logging.JsonLogFormatter* Python logging formatter that produces messages following the JSON schema for a common application logging format defined by the illustrious Mozilla Cloud Services group.

Configuration

Example configuration:

```
import logging.config

cfg = {
    'version': 1,
    'formatters': {
        'mozlog': {
            '()': 'dockerflow.logging.JsonLogFormatter',
            'logger_name': 'MyServiceName'
        }
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'mozlog'
        },
    },
    'loggers': {
        'myservice': {
            'handlers': ['console'],
            'level': 'DEBUG',
```

```
        },
    }
}

logging.config.dictConfig(cfg)

logging.info('I am logging in mozlog format now! woo hoo!')
```

In this example, we set up a logger for `myservice` (you'd replace that with your service name) which has a single handler named `console` which uses the `mozlog` formatter to output log event data to stdout.

API

class `dockerflow.logging.`**`JsonLogFormatter`**(*format=None*, *datefmt=None*, *logger_name='TestPilot'*)

Log formatter that outputs machine-readable json.

This log formatter outputs JSON format messages that are compatible with Mozilla's standard heka-based log aggregation infrastructure.

See also: https://mana.mozilla.org/wiki/display/CLOUDSERVICES/Logging+Standard https://mana.mozilla.org/wiki/pages/viewpage.action?pageId=42895640

Adapted from: https://github.com/mozilla-services/mozservices/blob/master/mozsvc/util.py#L106

Django

The package `dockerflow.django` package implements various tools to support Django projects that want to follow the Dockerflow specs:

- A Python logging formatter following the mozlog format to be used in the `LOGGING` setting.
- A middleware to emit request.summary log records based on request specific data.
- Views for health monitoring:
  - `/__version__` - Serves a `version.json` file
  - `/__heartbeat__` - Run Django checks as configured in the `DOCKERFLOW_CHECKS` setting
  - `/__lbheartbeat__` - Retuns a HTTP 200 response
- Signals for passed and failed heartbeats.

Setup

To install `python-dockerflow`'s Django support please follow these steps:

1. Add `dockerflow.django` to your `INSTALLED_APPS` setting
2. Define a `BASE_DIR` setting that is the root path of your Django project. This will be used to locate the `version.json` file that is generated by CircleCI or another process during deployment.

   See also:

   Versions for more information
3. Add the `DockerflowMiddleware` to your `MIDDLEWARE_CLASSES` or `MIDDLEWARE` setting:

```
MIDDLEWARE_CLASSES = (
    # ...
    'dockerflow.django.middleware.DockerflowMiddleware',
    # ...
)
```

4. Configure logging to use the `JsonLogFormatter` logging formatter for the `request.summary` logger (you may have to extend your existing logging configuration):

```
LOGGING = {
    'version': 1,
    'formatters': {
        'json': {
            '()': 'dockerflow.logging.JsonLogFormatter',
            'logger_name': '<< MySiteName >>'
        }
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'json'
        },
    },
    'loggers': {
        'request.summary': {
            'handlers': ['console'],
            'level': 'DEBUG',
        },
    }
}
```

See also:

Logging for more information

Configuration

Accept its configuration through environment variables.

There are several options to handle configuration values through environment variables, e.g. as shown in the configuration grid on djangopackages.com.

os.environ

The simplest is to use Pyhton's `os.environ` object to access environment variables for settings and other variables, e.g.:

```
MY_SETTING = os.environ.get('DJANGO_MY_SETTING', 'default value')
```

The downside of that is that it nicely works only for string based variables, since that's what `os.environ` returns.

python-decouple

A good replacement is python-decouple as it's agnostic to the framework in use and offers casting the returned value to the type wanted, e.g.:

```
from decouple import config

MY_SETTING = config('DJANGO_MY_SETTING', default='default value')
DEBUG = config('DJANGO_DEBUG', default=False, cast=bool)
```

As you can see the `DEBUG` setting would be populated from the `DJANGO_DEBUG` environment variable but also be cast as a boolean (while considering the string values `'1'`, `'yes'`, `'true'` and `'on'` as truthy values, and similar for falsey values).

django-environ

Django-environ follows similar patterns as python-decouple but implements specific casters for typical Django settings. E.g.:

```
import environ
env = environ.Env()

MY_SETTING = env.str('DJANGO_MY_SETTING', default='default value')
DEBUG = env.bool('DJANGO_DEBUG', default=False)
DATABASES = {
    'default': env.db(),  # automatically looks for DATABASE_URL
}
```

django-configurations

If you're interested in even more complex scenarios there are tools like django-configurations which allows loading different sets of settings depending on an additional environment variable `DJANGO_CONFIGURATION` to separate settings by environment (e.g. dev, stage, prod). It also ships with `Value` classes that implement configuration parsing from environment variable and casting, e.g.:

```
from configurations import Configuration, values

class Dev(Configuration):
    SESSION_COOKIE_SECURE = False
    DEBUG = values.BooleanValue(default=False)

class Prod(Dev):
    SESSION_COOKIE_SECURE = True
```

In that example the configuration class that is given in the `DJANGO_CONFIGURATION` environment variable would be used as the base for Django's settings.

PORT

Listen on environment variable `$PORT` for HTTP requests.

Depending on which WSGI server you are using to run your Python application there are different ways to accept the `PORT` as the port to launch your application with.

Gunicorn

Gunicorn automatically will bind to the hostname:port combination of `0.0.0.0:$PORT` if it find the `PORT` environment variable. That means running gunicorn is as simple as using this:

```
gunicorn <project>.wsgi:application --workers 4 --access-logfile -
```

See also:

The full gunicorn documentation for more details.

uWSGI

For uWSGI all you have to do is to bind on the `PORT` when you define the `uwsgi.ini`, e.g.:

```
[uwsgi]
http-socket = :$(PORT)
master = true
processes = 4
module = <project>.wsgi:application
chdir = /app
enable-threads = True
```

See also:

The full uWSGI documentation for more details.

Versions

> Must have a JSON version object at /app/version.json.

Dockerflow requires writing a version object to the file `/app/version.json` as see from the docker container to be served under the URL path `/__version__`.

To facilitate this python-dockerflow contains a Django view to read the file under path `BASE_DIR + 'version.json'` where `BASE_DIR` is required to be defined in the Django project settings, e.g.:

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

Assuming that the `settings.py` file is contained in the project folder That means the `BASE_DIR` setting will be the one where the `manage.py` file is located in the below example directory tree:

```
.
- .dockerignore
- .gitignore
- .travis.yml
- Dockerfile
- README.rst
- circle.yml
- manage.py
- requirements.txt
- staticfiles
|   - ..
- tests
|   - ..
- version.json
```

```
- <project>
|   - app1
|   |   - ..
|   |   - ..
|   - app2
|   |   - ..
|   |   - ..
|   - settings.py
|   - urls.py
- ..
```

Health

TODO

Logging

Dockerflow provides a *dockerflow.logging.JsonLogFormatter* Python logging formatter class.

To use it, put something like this in your Django `settings` file:

```
LOGGING = {
    'version': 1,
    'formatters': {
        'mozlog': {
            '()': 'dockerflow.logging.JsonLogFormatter',
            'logger_name': 'MyServiceName'
        }
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'mozlog'
        },
    },
    'loggers': {
        'myservice': {
            'handlers': ['console'],
            'level': 'DEBUG',
        },
    }
}
```

Static content

To properly serve static content it's recommended to use Whitenoise. It contains a middleware that is able to serve files that were built by Django's collectstatic management command (e.g. including bundle files built by django-pipeline) with **far-future headers** and proper response headers for the AWS CDN to work.

To enable Whitenoise, please install it from PyPI and then enable it in your Django projet:

1. Set your `STATIC_ROOT` setting:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

2. Add the middleware to your `MIDDLEWARE` (or `MIDDLEWARE_CLASSES`) setting:

```
MIDDLEWARE_CLASSES = [
    # 'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    # ...
]
```

   Make sure to follow the SecurityMiddleware.

3. Enable the staticfiles storage that is able to compress files during collection and ship them with far-future headers:

```
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

1. Install brotlipy so the storage can generate compressed files of your static files in the brotli format.

For more configuration options and details how to use Whitenoise see the section about Using WhiteNoise with Django in its documentation.

Settings

DOCKERFLOW_VERSION_CALLBACK

The dotted import path for the callable that returns the content to return under `/__version__`.

Defaults to `'dockerflow.version.get_version'` which will be passed the `BASE_DIR` setting by default.

DOCKERFLOW_CHECKS

A list of dotted import paths to register during Django setup, to be used in the rendering of the `/__heartbeat__` view. Defaults to:

```
DOCKERFLOW_CHECKS = [
    'dockerflow.django.checks.check_database_connected',
    'dockerflow.django.checks.check_migrations_applied',
]
```

API

This page shows the various code paths available in python-dockerflow.

Version

dockerflow.version.**get_version**(*root*)

Load and return the contents of version.json.

Parameters **root** (*str*) – The root path that the `version.json` file will be opened

Returns Content of `version.json` or None

Return type dict or None

Django

Checks

The provided checks hook into Django's system check framework to enable the `heartbeat view` to diagnose the current health of the Django project.

Signals

During the rendering of the `/__heartbeat__` view two signals are being sent to hook into the result of the checks:

`dockerflow.django.signals.`**`heartbeat_passed`**
The signal that is sent when the heartbeat checks pass successfully.

`dockerflow.django.signals.`**`heartbeat_failed`**
The signal that is sent when the heartbeat checks raise either a warning or worse (error, critical)

Both signals receive an additional `level` parameter that indicates the maximum check level that failed during the rendering.

E.g. to hook into those signals to send data to statsd, do this:

```
from django.dispatch import receiver
from dockerflow.django.signals import heartbeat_passed, heartbeat_failed
from statsd.defaults.django import statsd

@receiver(heartbeat_passed)
def heartbeat_passed_handler(sender, level, **kwargs):
    statsd.incr('heartbeat.pass')

@receiver(heartbeat_failed)
def heartbeat_failed_handler(sender, level, **kwargs):
    statsd.incr('heartbeat.fail')
```

Views

`dockerflow.django` implements various views so the automatic application monitoring can happen. They are mounted by including them in the root of a URL configration:

```
urlpatterns = [
    url(r'^', include('dockerflow.django.urls', namespace='dockerflow')),
    # ...
]
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index