python-dockerflow Documentation

Release 2019.10.0.post8

Mozilla Foundation

Jun 10, 2020

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*, *Flask*, *Sanic*

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

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

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*, *Flask*, *Sanic*

logging Send text logs to `stdout` or `stderr`. See: *Generic*, *Django*, *Flask*, *Sanic*

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

CHAPTER 5

Contents

5.1 Authors

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

5.2 Changelog

5.2.1 2020.06.0 (2020-06-09)

- Set heartbeat fail level to checks.ERROR

5.2.2 2019.10.0 (2019-10-28)

- Add Python 3.8 support.
- Fix a regression in the JSON logger parameter signature introduced in version 2018.2.1.
- Fixed some test harness issues, e.g. broken version contraint on the Django 2.2 tests.
- Speed up tests by only installing framework dependencies when needed.

5.2.3 2019.9.0 (2019-09-26)

- Make JsonLogFormatter easier to extend
- Blacken and isorted source code.

5.2.4 2019.6.0 (2019-06-25)

- Add support for Sanic 19.
- Add support for Python 3.7 and Django 2.1 and 2.2.
- Drop support for Python 3.4 and 3.5 and Django 1.8, 1.9, 1.10 and 2.0.
- Match Django urlpatterns with trailing slash.
- Use black for code formatting.

5.2.5 2019.5.0 (2019-05-13)

- Gracefully handle user loading to prevent accidental race condtions during exception handling when using the Flask Dockerflow extension.

5.2.6 2018.4.0 (2018-04-03)

- Fix backward-compatibility in the `check_migrations_applied` Flask check when an older version of Flask-Migrate is used.

5.2.7 2018.2.1 (2018-02-24)

- Fixes the instantiation of the JsonLogFormatter logging formatter on Python 3 when using the logging module's ability to be configured with ConfigParser ini files.
- Extend the documentation for custom checks and reorganized it a bit.

5.2.8 2018.2.0 (2018-02-20)

- Adds Flask support. See the documentation for more information.
- Extends the documentation about defining custom health checks.
- Refactored some of the health monitoring code that existed for the Django support.
- Fixed an embarrassing typo about the default logger name when using the `JsonLogFormatter` logging formatter, changed it `TestPilot` to `Dockerflow`.
- Extends the testing matrix to include Django 2.0.
- Make sure the the combination of Python and Django versions match the official recommendation as defined at https://docs.djangoproject.com/en/2.0/faq/install/#what-python-version-can-i-use-with-django.

5.2.9 2017.11.0 (2017-11-16)

- Fixed name of mozlog message field from "message" to "msg" as specified in https://wiki.mozilla.org/Firefox/Services/Logging. Thanks @leplatrem!

5.2.10 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.

5.2.11 2017.4.0 (2017-04-09)

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

5.2.12 2017.1.1 (2017-01-25)

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

5.2.13 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.

5.2.14 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

5.3 Logging

python-dockerflow provides a *`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.

See also:

For more information see the *API documentation* for the `dockerflow.logging` module.

5.3.1 Configuration

There a different ways to configure Python logging, please refer to the `logging` documentation to learn more.

The following examples should be considered excerpts and won't be enough for your application to work. They only illustrate how to use the JSON logging formatter for a specific logger.

Dictionary based

A simple example configuration for a `myproject` logger could look like this:

```
import logging.config

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

logging.config.dictConfig(cfg)
logger = logging.getLogger('myproject')
logger.info('I am logging in mozlog format now! woo hoo!')
```

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

ConfigParser ini file based

Consider an `ini` file with the following content that does the same thing as the dictionary based configuratio above:

Listing 1: logging.ini

```
[loggers]
keys = root, myproject

[handlers]
keys = console

[formatters]
```

(continues on next page)

(continued from previous page)

```
keys = json

[logger_root]
level = INFO
handlers = console

[logger_myproject]
level = DEBUG
handlers = console
qualname = myproject

[handler_console]
class = StreamHandler
level = DEBUG
args = (sys.stdout,)
formatter = json

[formatter_json]
class = dockerflow.logging.JsonLogFormatter
```

Then load the ini file using the `logging` module function `logging.config.fileConfig()`:

Listing 2: myproject.py

```
logging.config.fileConfig('logging.ini')
logger = logging.getLogger('myproject')
logger.info('I am logging in mozlog format now! woo hoo!')
```

5.4 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.

See also:

For more information see the *API documentation* for the `dockerflow.django` module.

5.4.1 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!).

5.4.2 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 Python'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.

5.4.3 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.

It's recommended to use port `8000` by default.

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 myproject.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 = myproject.wsgi:application
chdir = /app
enable-threads = True
```

See also:

The full uWSGI documentation for more details.

5.4.4 Versions

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

Dockerflow requires writing a version object to the file `/app/version.json` as seen 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
├── myproject
│   ├── app1
│   │   ├── ..
│   │   └── ..
│   ├── app2
│   │   ├── ..
│   │   └── ..
│   ├── settings.py
│   └── urls.py
└── ..
```

5.4.5 Health monitoring

Health monitoring happens via three different views following the Dockerflow spec:

GET /__version__
The view that serves the *version information*.

Example request:

```
GET /__version__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: application/json

{
  "commit": "52ce614fbf99540a1bf6228e36be6cef63b4d73b",
  "version": "2017.11.0",
  "source": "https://github.com/mozilla/telemetry-analysis-service",
  "build": "https://circleci.com/gh/mozilla/telemetry-analysis-service/2223"
}
```

Status Codes

- 200 OK – no error
- 404 Not Found – a version.json wasn't found

GET /__heartbeat__
The heartbeat view will go through the list of configured Dockerflow checks in the *DOCKERFLOW_CHECKS* setting, run each check and add their results to a JSON response.

The view will return HTTP responses with either an status code of 200 if all checks ran successfully or 500 if there was one or more warnings or errors returned by the checks.

Custom Dockerflow checks:

To write your own custom Dockerflow checks, please follow the documentation about `Django's system check framework` and particularly the section **"Writing your own checks"**.

Note: Don't forget to add the check additionally to the *DOCKERFLOW_CHECKS* setting once you've added it to your code.

Example request:

```
GET /__heartbeat__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 500 Internal Server Error
Vary: Accept-Encoding
Content-Type: application/json

{
  "status": "warning",
  "checks": {
    "check_debug": "ok",
```

(continues on next page)

(continued from previous page)

```
    "check_sts_preload": "warning"
  },
  "details": {
    "check_sts_preload": {
      "status": "warning",
      "level": 30,
      "messages": {
        "security.W021": "You have not set the SECURE_HSTS_PRELOAD setting to␣
↪True. Without this, your site cannot be submitted to the browser preload list."
      }
    }
  }
}
```

Status Codes

- 200 OK – no error
- 500 Internal Server Error – there was a warning or error

GET /__lbheartbeat__

The view that simply returns a successful HTTP response so that a load balancer in front of the application can check that the web application has started up.

Example request:

```
GET /__lbheartbeat__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: application/json
```

Status Codes

- 200 OK – no error

5.4.6 Logging

Dockerflow provides a *JsonLogFormatter* Python logging formatter class.

To use it, put something like this in your Django `settings` file and configure **at least** the `request.summary` logger that way:

```
LOGGING = {
    'version': 1,
    'formatters': {
        'json': {
            '()': 'dockerflow.logging.JsonLogFormatter',
            'logger_name': 'myproject'
        }
    },
    'handlers': {
```

(continues on next page)

(continued from previous page)

```
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'json'
        },
    },
    'loggers': {
        'request.summary': {
            'handlers': ['console'],
            'level': 'DEBUG',
        },
    }
}
```

5.4.7 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.

5.4.8 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',
]
```

5.5 Flask

The package `dockerflow.flask` package implements various tools to support Flask based projects that want to follow the Dockerflow specs:

- A Python logging formatter following the mozlog format.
- A Flask extension implements:
  - Emitting of request.summary log records based on request specific data.
  - Views for health monitoring:
    * `/__version__` - Serves a `version.json` file
    * `/__heartbeat__` - Runs the configured Dockerflow checks
    * `/__lbheartbeat__` - Retuns a HTTP 200 response
  - Signals for passed and failed heartbeats.
  - Built-in Dockerflow checks for SQLAlchemy and Redis connections and validating Alembic migrations.
  - Hooks to add custom Dockerflow checks.

See also:

For more information see the *API documentation* for the `dockerflow.flask` module.

5.5.1 Setup

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

1. In your code where your Flask application lives set up the dockerflow Flask extension:

   ```
   from flask import Flask
   from dockerflow.flask import Dockerflow

   app = Flask(__name__)
   dockerflow = Dockerflow(app)
   ```

2. Make sure the app root path is set correctly as 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. Configure logging to use the `JsonLogFormatter` logging formatter for the `request.summary` logger (you may have to extend your existing logging configuration), see *Logging* for more information.

5.5.2 Configuration

Accept its configuration through environment variables.

There are several options to handle configuration values through environment variables when configuring Flask.

os.environ

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

```
MY_SETTING = os.environ.get('FLASK_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('FLASK_MY_SETTING', default='default value')
DEBUG = config('FLASK_DEBUG', default=False, cast=bool)
```

As you can see the `DEBUG` configuration value would be populated from the `FLASK_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).

flask-environ

flask-environ follows similar patterns as python-decouple but implements specific casters for typical Flask configuration values. E.g.:

```
from flask import Flask
from flask_environ import get, collect, word_for_true

app = Flask(__name__)

app.config.update(collect(
    get('DEBUG', default=False, convert=word_for_true),
    get('HOST', default='127.0.0.1'),
    get('PORT', default=5000, convert=int),
    get('SECRET_KEY',
        'SQLALCHEMY_DATABASE_URI',
        'TWITTER_CONSUMER_KEY',
        'TWITTER_CONSUMER_SECRET',
    ),
))
```

Flask-Env

If you need to solve more complex configuration scenarios there are tools like Flask-Env which allows loading settings for different environments (e.g. dev, stage, prod) via environment variables. It provides a small Python meta class to allow setting up the configuration values:

E.g. in a `config.py` file next to your application:

```
from flask_env import MetaFlaskEnv

class Dev(metaclass=MetaFlaskEnv):
    DEBUG = True
    PORT = 5000

class Prod(Dev):
    DEBUG = False
```

Then in your application code:

```
import os
from flask import Flask

app = Flask(__name__)
app.config.from_object(os.environ.get('FLASK_CONFIG', 'config.Dev'))
```

In that example the configuration class that is given in the `FLASK_CONFIG` environment variable would be used to update the default Flask configuration values while allowing to override the values via environment variables.

It's recommended to use the Flask-Env feature to define a prefix for the environment variable it uses to check, e.g.:

```
from flask_env import MetaFlaskEnv

class Dev(metaclass=MetaFlaskEnv):
    ENV_PREFIX = 'ACME_'
    DEBUG = True
```

To override the config value of `DEBUG` the environment variable would be called `ACME_DEBUG`.

5.5.3 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.

It's recommended to use port `8000` by default.

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, for example:

```
gunicorn myproject:app --workers 4
```

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 = myproject:app
chdir = /app
enable-threads = True
```

See also:

The full uWSGI documentation for more details.

5.5.4 Versions

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

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

To facilitate this python-dockerflow comes with a Flask view to read the file under path the parent directory of the Flask app root. See the `Flask API docs` for more information about the app root path.

If you'd like to override the location from which the view is reading the `version.json` file from, simply override the optional `version_path` parameter to the *`Dockerflow`* class, e.g.:

```
from flask import Flask
from dockerflow.flask import Dockerflow

app = Flask(__name__)
dockerflow = Dockerflow(app, version_path='/app')
```

Alternatively if you'd like to completely override the way the version information is read use the *`version_callback()`* decorator to decorate a callback that gets the `version_path` value passed. E.g.:

```
import json
from flask import Flask
from dockerflow.flask import Dockerflow

app = Flask(__name__)
dockerflow = Dockerflow(app)

@dockerflow.version_callback
def my_version(root):
    return json.loads(os.path.join(root, 'acme_version.json'))
```

5.5.5 Health monitoring

Health monitoring happens via three different views following the Dockerflow spec:

`GET /__version__`
The view that serves the *version information*.

Example request:

```
GET /__version__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: application/json

{
  "commit": "52ce614fbf99540a1bf6228e36be6cef63b4d73b",
  "version": "2017.11.0",
  "source": "https://github.com/mozilla/telemetry-analysis-service",
  "build": "https://circleci.com/gh/mozilla/telemetry-analysis-service/2223"
}
```

Status Codes

- 200 OK – no error
- 404 Not Found – a version.json wasn't found

GET /__heartbeat__

The heartbeat view will go through the list of registered Dockerflow checks, run each check and add their results to a JSON response.

The view will return HTTP responses with either an status code of 200 if all checks ran successfully or 500 if there was one or more warnings or errors returned by the checks.

Built-in Dockerflow checks:

There are a few built-in checks that are automatically added to the list of checks if the appropriate Flask extension objects are passed to the *Dockerflow* class during instantiation.

For detailed examples please see the API documentation for the built-in *Flask Dockerflow checks*.

Custom Dockerflow checks:

To write your own custom Dockerflow checks simply write a function that returns a list of one or many check message instances representing the severity of the check result. The `dockerflow.flask.checks` module contains a series of predefined check messages for the severity levels: `Debug`, `Info`, `Warning`, `Error`, `Critical`.

Here's an example of a check that handles various levels of exceptions from an external storage system with different check message:

```
from dockerflow.flask import checks, Dockerflow

app = Flask(__name__)
dockerflow = Dockerflow(app)

@dockerflow.check
def storage_reachable():
    result = []
    try:
        acme.storage.ping()
    except SlowConnectionException as exc:
        result.append(checks.Warning(exc.msg, id='acme.health.0002'))
    except StorageException as exc:
```

(continues on next page)

(continued from previous page)

```
        result.append(checks.Error(exc.msg, id='acme.health.0001'))
    return result
```

Notice the use of the *check()* decorator to mark the check to be used.

Example request:

```
GET /__heartbeat__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 500 Internal Server Error
Vary: Accept-Encoding
Content-Type: application/json

{
  "status": "warning",
  "checks": {
    "check_debug": "ok",
    "check_sts_preload": "warning"
  },
  "details": {
    "check_sts_preload": {
      "status": "warning",
      "level": 30,
      "messages": {
        "security.W021": "You have not set the SECURE_HSTS_PRELOAD setting to
↪True. Without this, your site cannot be submitted to the browser preload list."
      }
    }
  }
}
```

Status Codes

- 200 OK – no error
- 500 Internal Server Error – there was a warning or error

GET /__lbheartbeat__

The view that simply returns a successful HTTP response so that a load balancer in front of the application can check that the web application has started up.

Example request:

```
GET /__lbheartbeat__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: application/json
```

Status Codes

- 200 OK – no error

5.5.6 Logging

Dockerflow provides a *JsonLogFormatter* Python logging formatter class.

To use it, put something like this **BEFORE** your Flask app is initialized for at least the `request.summary` logger:

```
from logging.conf import dictConfig

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

5.5.7 Static content

To properly serve static content it's recommended to use Whitenoise. It contains a WSGI middleware that is able to serve the files that Flask usually serves under the static URL path (Flask app parameter `static_url_path`) from the Flask app's static folder (`static_folder`) but with **far-future headers** and proper response headers for the CDNs.

For more information see the documentation dedicated to using Whitenoise with Flask.

Another great adition (especially if no JavaScript based build system is used like webpack) is using Flask-Assets, a Flask extension based on the webassets management tool. Since it also uses the Flask app's static folder as the output directory by default both work well together.

5.6 Sanic

The package `dockerflow.sanic` package implements various tools to support Sanic based projects that want to follow the Dockerflow specs:

- A Python logging formatter following the mozlog format.
- A Sanic extension implements:
  - Emitting of request.summary log records based on request specific data.
  - Views for health monitoring:

    * /__version__ - Serves a version.json file
    * /__heartbeat__ - Runs the configured Dockerflow checks
    * /__lbheartbeat__ - Retuns a HTTP 200 response
  - Signals for passed and failed heartbeats.
  - Built-in Dockerflow checks for SQLAlchemy and Redis connections and validating Alembic migrations.
  - Hooks to add custom Dockerflow checks.

See also:

For more information see the *API documentation* for the dockerflow.sanic module.

5.6.1 Setup

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

1. In your code where your Sanic application lives set up the dockerflow Sanic extension:

```
from sanic import Sanic
from dockerflow.sanic import Dockerflow

app = Sanic(__name__)
dockerflow = Dockerflow(app)
```

2. Make sure the app root path is set correctly as 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. Configure logging to use the JsonLogFormatter logging formatter for the request.summary logger (you may have to extend your existing logging configuration), see *Logging* for more information.

5.6.2 Configuration

> Accept its configuration through environment variables.

There are several options to handle configuration values through environment variables when configuring Sanic.

Sanic configuration

The simplest is to use Sanic's own ability to access environment variables for settings and other variables.

> Any variables defined with the SANIC_ prefix will be applied to the sanic config. For example, setting SANIC_REQUEST_TIMEOUT will be loaded by the application automatically and fed into the REQUEST_TIMEOUT config variable.
>
> —Sanic docs on configuration.

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('SANIC_MY_SETTING', default='default value')
DEBUG = config('SANIC_DEBUG', default=False, cast=bool)
```

As you can see the `DEBUG` configuration value would be populated from the `SANIC_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).

sanic-envconfig

If you need to solve more complex configuration scenarios there are tools like sanic-envconfig which allows loading settings for different environments (e.g. dev, stage, prod) via environment variables. It provides a small Python base class to allow setting up the configuration values:

E.g. in a `config.py` file next to your application:

```
from sanic_envconfig import EnvConfig

class Dev(EnvConfig):
    DEBUG: bool = True
    DB_URL: str = None
    WORKERS: int = 1
    PORT: int = 5000

class Prod(Dev):
    DEBUG: bool = False
```

Then in your application code:

```
import os
from sanic import Sanic

app = Sanic(__name__)
app.config.from_object(os.environ.get('SANIC_CONFIG', 'config.Dev'))
```

In that example the configuration class that is given in the `SANIC_CONFIG` environment variable would be used to update the default Sanic configuration values while allowing to override the values via environment variables.

It's recommended to use the sanic-envconfig feature to define a prefix for the environment variable it uses to check, e.g.:

```
from sanic_envconfig import EnvConfig

class Dev(EnvConfig):
    _ENV_PREFIX = 'ACME_'
    DEBUG = True
```

To override the config value of `DEBUG` the environment variable would be called `ACME_DEBUG`.

5.6.3 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.

It's recommended to use port `8000` by default.

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, for example:

```
gunicorn myproject:app --worker-class sanic.worker.GunicornWorker
```

See also:

The full gunicorn documentation for more details.

ASGI

Sanic is also ASGI-compliant. This means you can use your preferred ASGI webserver to run Sanic. The three main implementations of ASGI are Daphne, Uvicorn, and Hypercorn.

See also:

The Sanic deployment documentation has more for more details.

5.6.4 Versions

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

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

To facilitate this python-dockerflow comes with a Sanic view to read the file under the current worked directory (`.`).

If you'd like to override the location from which the view is reading the `version.json` file from, simply override the optional `version_path` parameter to the *`Dockerflow`* class, e.g.:

```
from sanic import Sanic
from dockerflow.sanic import Dockerflow

app = Sanic(__name__)
dockerflow = Dockerflow(app, version_path='/app')
```

Alternatively if you'd like to completely override the way the version information is read use the *`version_callback()`* decorator to decorate a callback that gets the `version_path` value passed. E.g.:

```
import json
from sanic import Sanic
from dockerflow.sanic import Dockerflow

app = Sanic(__name__)
dockerflow = Dockerflow(app)
```

(continues on next page)

(continued from previous page)

```
@dockerflow.version_callback
def my_version(root):
    return json.loads(os.path.join(root, 'acme_version.json'))
```

5.6.5 Health monitoring

Health monitoring happens via three different views following the Dockerflow spec:

GET /__version__
The view that serves the *version information*.

Example request:

```
GET /__version__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: application/json

{
  "commit": "52ce614fbf99540a1bf6228e36be6cef63b4d73b",
  "version": "2017.11.0",
  "source": "https://github.com/mozilla/telemetry-analysis-service",
  "build": "https://circleci.com/gh/mozilla/telemetry-analysis-service/2223"
}
```

Status Codes

- 200 OK – no error
- 404 Not Found – a version.json wasn't found

GET /__heartbeat__
The heartbeat view will go through the list of registered Dockerflow checks, run each check and add their results to a JSON response.

The view will return HTTP responses with either an status code of 200 if all checks ran successfully or 500 if there was one or more warnings or errors returned by the checks.

Built-in Dockerflow checks:

There are a few built-in checks that are automatically added to the list of checks if the appropriate Sanic extension objects are passed to the *`Dockerflow`* class during instantiation.

For detailed examples please see the API documentation for the built-in *Sanic Dockerflow checks*.

Custom Dockerflow checks:

To write your own custom Dockerflow checks simply write a function that returns a list of one or many check message instances representing the severity of the check result. The `dockerflow.sanic.checks` module contains a series of predefined check messages for the severity levels: `Debug`, `Info`, `Warning`, `Error`, `Critical`.

Here's an example of a check that handles various levels of exceptions from an external storage system with different check message:

```
   from sanic import Sanic
   from dockerflow.sanic import checks, Dockerflow

   app = Sanic(__name__)
   dockerflow = Dockerflow(app)

   @dockerflow.check
   async def storage_reachable():
       result = []
       try:
           acme.storage.ping()
       except SlowConnectionException as exc:
           result.append(checks.Warning(exc.msg, id='acme.health.0002'))
       except StorageException as exc:
           result.append(checks.Error(exc.msg, id='acme.health.0001'))
       return result

also works without async::

   @dockerflow.check
       def storage_reachable():
           result = []
           # ...
```

Notice the use of the *check()* decorator to mark the check to be used.

Example request:

```
GET /__heartbeat__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 500 Internal Server Error
Vary: Accept-Encoding
Content-Type: application/json

{
  "status": "warning",
  "checks": {
    "check_debug": "ok",
    "check_sts_preload": "warning"
  },
  "details": {
    "check_sts_preload": {
      "status": "warning",
      "level": 30,
      "messages": {
        "security.W021": "You have not set the SECURE_HSTS_PRELOAD setting to␣
↪True. Without this, your site cannot be submitted to the browser preload list."
      }
    }
  }
}
```

Status Codes

- 200 OK – no error
- 500 Internal Server Error – there was a warning or error

GET /__lbheartbeat__

The view that simply returns a successful HTTP response so that a load balancer in front of the application can check that the web application has started up.

Example request:

```
GET /__lbheartbeat__ HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: application/json
```

Status Codes

- 200 OK – no error

5.6.6 Logging

Dockerflow provides a *JsonLogFormatter* Python logging formatter class.

To use it, pass something like this to your Sanic app when it is initialized for at least the `request.summary` logger:

```
from sanic import Sanic

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

sanic = Sanic(__name__, log_config=log)
```

By default the `log_info` parameter has the value of `sanic.log.LOGGING_CONFIG_DEFAULTS`.

Alternatively you can also pass the same logging config dictionary to the `logging.conf.dictConfig` utility **BEFORE** your Sanic app is initialized:

```
from logging.conf import dictConfig
from sanic import Sanic

log_config = {
    # ...
}

dictConfig(log_config)

sanic = Sanic(__name__)
```

5.6.7 Static content

Please refer to the Sanic documentation about serving static files for more information.

5.7 API

This section shows more details about the following code paths available in python-dockerflow:

5.7.1 Django

This documents the code that allows Django integration.

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.

`dockerflow.django.checks.`**`check_database_connected`**(*app_configs*, ***kwargs*)
A Django check to see if connecting to the configured default database backend succeeds.

`dockerflow.django.checks.`**`check_migrations_applied`**(*app_configs*, ***kwargs*)
A Django check to see if all migrations have been applied correctly.

`dockerflow.django.checks.`**`check_redis_connected`**(*app_configs*, ***kwargs*)
A Django check to connect to the default redis connection using `django_redis.get_redis_connection` and see if Redis responds to a `PING` command.

Signals

During the rendering of the `/__heartbeat__` Django 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')),
    # ...
]
```

`dockerflow.django.views.`**`heartbeat`**(*request*)
Runs all the Django checks and returns a JsonResponse with either a status code of 200 or 500 depending on the results of the checks.

Any check that returns a warning or worse (error, critical) will return a 500 response.

`dockerflow.django.views.`**`lbheartbeat`**(*request*)
Let the load balancer know the application is running and available must return 200 (not 204) for ELB http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/elb-healthchecks.html

`dockerflow.django.views.`**`version`**(*request*)
Returns the contents of version.json or a 404.

5.7.2 Flask

This documents the various Flask specific functionality but doesn't cover internals of the extension.

Extension

class `dockerflow.flask.app.`**`Dockerflow`**(*app=None*, *db=None*, *redis=None*, *migrate=None*, *silenced_checks=None*, *version_path=None*, **args*, ***kwargs*)
The Dockerflow Flask extension. Set it up like this:

Listing 3: `myproject.py`

```
from flask import Flask
from dockerflow.flask import Dockerflow

app = Flask(__name__)
dockerflow = Dockerflow(app)
```

Or if you use the Flask application factory pattern, in an own module set up Dockerflow first:

Listing 4: `myproject/deployment.py`

```
from dockerflow.flask import Dockerflow

dockerflow = Dockerflow()
```

and then import and initialize it with the Flask application object when you create the application:

Listing 5: `myproject/app.py`

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from myproject.deployment import dockerflow
    dockerflow.init_app(app)

    from myproject.views.admin import admin
    from myproject.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)

     return app
```

See the parameters for a more detailed list of optional features when initializing the extension.

Parameters

- **app** (*Flask or None*) – The Flask app that this Dockerflow extension should be initialized with.
- **db** – A Flask-SQLAlchemy extension instance to be used by the built-in Dockerflow check for the database connection.
- **redis** – A Redis connection to be used by the built-in Dockerflow check for the Redis connection.
- **migrate** – A Flask-Migrate extension instance to be used by the built-in Dockerflow check for Alembic migrations.
- **silenced_checks** (*list*) – Dockerflow check IDs to ignore when running through the list of configured checks.
- **version_path** – The filesystem path where the `version.json` can be found. Defaults to the parent directory of the Flask app's root path.

check (*func=None*, *name=None*)

A decorator to register a new Dockerflow check to be run when the /__heartbeat__ endpoint is called., e.g.:

```
from dockerflow.flask import checks

@dockerflow.check
def storage_reachable():
    try:
        acme.storage.ping()
    except SlowConnectionException as exc:
        return [checks.Warning(exc.msg, id='acme.health.0002')]
    except StorageException as exc:
        return [checks.Error(exc.msg, id='acme.health.0001')]
```

or using a custom name:

```
@dockerflow.check(name='acme-storage-check)
def storage_reachable():
    # ...
```

init_app(*app*)
Initializes the extension with the given app, registers the built-in views with an own blueprint and hooks up our signal callbacks.

init_check(*check*, *obj*)
Adds a given check callback with the provided object to the list of checks. Useful for built-ins but also advanced custom checks.

summary_extra()
Build the extra data for the summary logger.

user_id()
Return the ID of the current request's user

version_callback(*func*)
A decorator to optionally register a new Dockerflow version callback and use that instead of the default of *dockerflow.version.get_version()*.

The callback will be passed the value of the `version_path` parameter to the Dockerflow extension object, which defaults to the parent directory of the Flask app's root path.

The callback should return a dictionary with the version information as defined in the Dockerflow spec, or None if no version information could be loaded.

E.g.:

```
app = Flask(__name__)
dockerflow = Dockerflow(app)

@dockerflow.version_callback
def my_version(root):
    return json.loads(os.path.join(root, 'acme_version.json'))
```

exception dockerflow.flask.app.**HeartbeatFailure**(*description=None*, *response=None*, *original_exception=None*)

Checks

dockerflow.flask.checks.**check_database_connected**(*db*)
A built-in check to see if connecting to the configured default database backend succeeds.

It's automatically added to the list of Dockerflow checks if a `SQLAlchemy` object is passed to the *Dockerflow* class during instantiation, e.g.:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from dockerflow.flask import Dockerflow

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)

dockerflow = Dockerflow(app, db=db)
```

dockerflow.flask.checks.**check_migrations_applied**(*migrate*)

A built-in check to see if all migrations have been applied correctly.

It's automatically added to the list of Dockerflow checks if a flask_migrate.Migrate object is passed to the *Dockerflow* class during instantiation, e.g.:

```
from flask import Flask
from flask_migrate import Migrate
from flask_sqlalchemy import SQLAlchemy
from dockerflow.flask import Dockerflow

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)
migrate = Migrate(app, db)

dockerflow = Dockerflow(app, db=db, migrate=migrate)
```

dockerflow.flask.checks.**check_redis_connected**(*client*)

A built-in check to connect to Redis using the given client and see if it responds to the `PING` command.

It's automatically added to the list of Dockerflow checks if a `StrictRedis` instances is passed to the *Dockerflow* class during instantiation, e.g.:

```
import redis
from flask import Flask
from dockerflow.flask import Dockerflow

app = Flask(__name__)
redis_client = redis.StrictRedis(host='localhost', port=6379, db=0)

dockerflow = Dockerflow(app, redis=redis)
```

An alternative approach to instantiating a Redis client directly would be using the Flask-Redis Flask extension:

```
from flask import Flask
from flask_redis import FlaskRedis
from dockerflow.flask import Dockerflow

app = Flask(__name__)
app.config['REDIS_URL'] = 'redis://:password@localhost:6379/0'
redis_store = FlaskRedis(app)

dockerflow = Dockerflow(app, redis=redis_store)
```

Signals

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

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

`dockerflow.flask.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 dockerflow.flask.signals import heartbeat_passed, heartbeat_failed
from myproject.stats import statsd

@heartbeat_passed.connect_via(app)
def heartbeat_passed_handler(sender, level, **extra):
    statsd.incr('heartbeat.pass')

@heartbeat_failed.connect_via(app)
def heartbeat_failed_handler(sender, level, **extra):
    statsd.incr('heartbeat.fail')
```

5.7.3 Logging

Generic tools for Python logging integration.

class `dockerflow.logging.`**`JsonLogFormatter`**(*fmt=None*, *datefmt=None*, *style='%'*, *logger_name='Dockerflow'*)

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://wiki.mozilla.org/Firefox/Services/Logging

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

`convert_record`(*record*)
Convert a Python LogRecord attribute into a dict that follows MozLog application logging standard.

- from - https://docs.python.org/3/library/logging.html#logrecord-attributes
- to - https://wiki.mozilla.org/Firefox/Services/Logging

`format`(*record*)
Format a Python LogRecord into a JSON string following MozLog application logging standard.

`is_value_jsonlike`(*value*)
Return True if the value looks like JSON. Use only on strings.

class dockerflow.logging.**SafeJSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

dockerflow.logging.**safer_format_traceback**(*exc_typ*, *exc_val*, *exc_tb*)

Format an exception traceback into safer string. We don't want to let users write arbitrary data into our logfiles, which could happen if they e.g. managed to trigger a ValueError with a carefully-crafted payload. This function formats the traceback using "%r" for the actual exception data, which passes it through repr() so that any special chars are safely escaped.

5.7.4 Sanic

This documents the various Sanic specific functionality but doesn't cover internals of the extension.

Extension

class dockerflow.sanic.app.**Dockerflow**(*app=None*, *redis=None*, *silenced_checks=None*, *version_path='.'*, **args*, ***kwargs*)

The Dockerflow Sanic extension. Set it up like this:

Listing 6: `myproject.py`

```
from sanic import Sanic
from dockerflow.sanic import Dockerflow

app = Sanic(__name__)
dockerflow = Dockerflow(app)
```

Or if you use the Sanic application factory pattern, in an own module set up Dockerflow first:

Listing 7: `myproject/deployment.py`

```
from dockerflow.sanic import Dockerflow

dockerflow = Dockerflow()
```

and then import and initialize it with the Sanic application object when you create the application:

Listing 8: `myproject/app.py`

```
def create_app(config_filename):
    app = Sanic(__name__)
    app.config.from_pyfile(config_filename)

    from myproject.deployment import dockerflow
    dockerflow.init_app(app)

    from myproject.views.admin import admin
    from myproject.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)

    return app
```

See the parameters for a more detailed list of optional features when initializing the extension.

Parameters

- **app** (*Sanic or None*) – The Sanic app that this Dockerflow extension should be initialized with.
- **redis** – A SanicRedis instance to be used by the built-in Dockerflow check for the sanic_redis connection.
- **silenced_checks** (*list*) – Dockerflow check IDs to ignore when running through the list of configured checks.
- **version_path** – The filesystem path where the `version.json` can be found. Defaults to `.`.

check(*func=None*, *name=None*)

A decorator to register a new Dockerflow check to be run when the /__heartbeat__ endpoint is called., e.g.:

```
from dockerflow.sanic import checks

@dockerflow.check
async def storage_reachable():
    try:
        acme.storage.ping()
    except SlowConnectionException as exc:
        return [checks.Warning(exc.msg, id='acme.health.0002')]
    except StorageException as exc:
        return [checks.Error(exc.msg, id='acme.health.0001')]
```

also works without async:

```
@dockerflow.check
def storage_reachable():
    # ...
```

or using a custom name:

```
@dockerflow.check(name='acme-storage-check')
async def storage_reachable():
    # ...
```

init_app(*app*)
Add the Dockerflow views and middleware to app.

init_check(*check*, *obj*)
Adds a given check callback with the provided object to the list of checks. Useful for built-ins but also advanced custom checks.

summary_extra(*request*)
Build the extra data for the summary logger.

version_callback(*func*)
A decorator to optionally register a new Dockerflow version callback and use that instead of the default of *dockerflow.version.get_version()*.

The callback will be passed the value of the `version_path` parameter to the Dockerflow extension object, which defaults to the parent directory of the Sanic app's root path.

The callback should return a dictionary with the version information as defined in the Dockerflow spec, or None if no version information could be loaded.

E.g.:

```
import aiofiles

app = Sanic(__name__)
dockerflow = Dockerflow(app)

@dockerflow.version_callback
async def my_version(root):
    path = os.path.join(root, 'acme_version.json')
    async with aiofiles.open(path, mode='r') as f:
        raw = await f.read()
    return json.loads(raw)
```

Checks

dockerflow.sanic.checks.**check_redis_connected**(*redis*)
A built-in check to connect to Redis using the given client and see if it responds to the `PING` command.

It's automatically added to the list of Dockerflow checks if a `SanicRedis` instance is passed to the *Dockerflow* class during instantiation, e.g.:

```
import redis as redislib
from sanic import Sanic
from dockerflow.sanic import Dockerflow

app = Sanic(__name__)
redis = redislib.from_url("redis://:password@localhost:6379/0")
dockerflow = Dockerflow(app, redis=redis)
```

An alternative approach to instantiating a Redis client directly would be using the Sanic-Redis Sanic extension:

```
from sanic import Sanic
from sanic_redis import SanicRedis
from dockerflow.sanic import Dockerflow

app = Sanic(__name__)
app.config['REDIS'] = {'address': 'redis://:password@localhost:6379/0'}
```

(continues on next page)

(continued from previous page)

```
redis = SanicRedis(app)
dockerflow = Dockerflow(app, redis=redis)
```

5.7.5 Version

Generic tools for version information.

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

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

HTTP Routing Table

/__heartbeat__

/__lbheartbeat__

/__version__

Index