Healthcheck is a library to write simple healthcheck functions that can be used to monitor your application. It is possible to use in a flask app or tornado app. It's useful for asserting that your dependencies are up and running and your application can respond to HTTP requests. The Healthcheck functions can be exposed via a user defined flask route so you can use an external monitoring application (Monit, nagios, Runscope, etc.) to check the status and uptime of your application.

New in version 1.1: Healthcheck also gives you a simple Flask route to view information about your application's environment. By default, this includes data about the operating system, the Python environment, the current process, and the application config. You can customize which sections are included, or add your own sections to the output.

Installing

pip install py-healthcheck

Usage

Here's an example of basic usage with Flask

```
from flask import Flask

from healthcheck import HealthCheck, EnvironmentDump


app = Flask(__name__)


health = HealthCheck()

envdump = EnvironmentDump()


# add your own check function to the healthcheck

def redis_available():

    client = _redis_client()

    info = client.info()

    return True, "redis ok"


health.add_check(redis_available)
```

```
# add your own data to the environment dump
def application_data():
    return {"maintainer": "Luis Fernando Gomes",
            "git_repo": "https://github.com/ateliedocodigo/py-healthcheck"}

envdump.add_section("application", application_data)

# Add a flask route to expose information
app.add_url_rule("/healthcheck", "healthcheck", view_func=lambda: health.run())
app.add_url_rule("/environment", "environment", view_func=lambda: envdump.run())
```

To use with Tornado you can import the Tornado Handler:

```
import tornado.web
from healthcheck import TornadoHandler, HealthCheck, EnvironmentDump

app = tornado.web.Application()

health = HealthCheck()
envdump = EnvironmentDump()

# add your own check function to the healthcheck
def redis_available():
    client = _redis_client()
    info = client.info()
    return True, "redis ok"

health.add_check(redis_available)

# add your own data to the environment dump or healthcheck
def application_data():
    return {"maintainer": "Luis Fernando Gomes",
```

```
        "git_repo": "https://github.com/ateliedocodigo/py-healthcheck"}

# ou choose where you want to output this information
health.add_section("application", application_data)
health.add_section("version", __version__)
envdump.add_section("application", application_data)

# Add a tornado handler to expose information
app.add_handlers(
    r".*",
    [
        (
            "/healthcheck",
            TornadoHandler, dict(checker=health)
        ),
        (
            "/environment",
            TornadoHandler, dict(checker=envdump)
        ),
    ]
)
```

Alternatively you can set all together:

```
import tornado.web
from healthcheck import TornadoHandler, HealthCheck, EnvironmentDump

# add your own check function to the healthcheck
def redis_available():
    client = _redis_client()
    info = client.info()
    return True, "redis ok"
```

```
health = HealthCheck(checkers=[redis_available])


# add your own data to the environment dump
def application_data():
    return {"maintainer": "Luis Fernando Gomes",
            "git_repo": "https://github.com/ateliedocodigo/py-healthcheck"}


envdump = EnvironmentDump(application=application_data)


app = tornado.web.Application([
    ("/healthcheck", TornadoHandler, dict(checker=health)),
    ("/environment", TornadoHandler, dict(checker=envdump)),
])
```

To run all of your check functions, make a request to the healthcheck URL you specified, like this:

curl http://localhost:5000/healthcheck

And to view the environment data, make a check to the URL you specified for EnvironmentDump:

curl http://localhost:5000/environment

The HealthCheck class

Check Functions

Check functions take no arguments and should return a tuple of (bool, str). The boolean is whether or not the check passed. The message is any string or output that should be rendered for this check. Useful for error messages/debugging.

```
# add check functions
def addition_works():
    if 1 + 1 == 2:
        return True, "addition works"
    else:
```

```
    return False, "the universe is broken"
```

Any exceptions that get thrown by your code will be caught and handled as errors in the healthcheck:

```
# add check functions
def throws_exception():
    bad_var = None
    bad_var['explode']
```

Will output:

```
{
    "status": "failure",
    "results": [
        {
            "output": "'NoneType' object has no attribute '__getitem__'",
            "checker": "throws_exception",
            "passed": false
        }
    ]
}
```

Note, all checkers will get run and all failures will be reported. It's intended that they are all separate checks and if any one fails the healthcheck overall is failed.

Caching

In Runscope's infrastructure, the /healthcheck endpoint is hit surprisingly often. haproxy runs on every server, and each haproxy hits every healthcheck twice a minute. (So if we have 30 servers in our infrastructure, that's 60 healthchecks per minute to every Flask service.) Plus, monit hits every healthcheck 6 times a minute.

To avoid putting too much strain on backend services, health check results can be cached in process memory. By default, health checks that succeed are cached for 27 seconds, and failures are cached for 9 seconds. These can be overridden with the success ttl and failed ttl parameters. If you don't want to use the cache at all, initialize the Healthcheck object with success_ttl=None, failed_ttl=None.

Customizing

You can customize the status codes, headers, and output format for success and failure responses.

The EnvironmentDump class

Built-in data sections

By default, EnvironmentDump data includes these 4 sections:

Os : information about your operating system.

Python : information about your Python executable, Python path, and installed packages.

process: information about the currently running Python process, including the PID, command line arguments, and all environment variables.

Some of the data is scrubbed to avoid accidentally exposing passwords or access keys/tokens. Config keys and environment variable names are scanned for key, token, or pass. If those strings are present in the name of the variable, the value is not included.

Disabling built-in data sections

For security reasons, you may want to disable an entire section. You can disable sections when you instantiate the EnvironmentDump object, like this:

```
envdump = EnvironmentDump(include_python=False,
                          include_os=False,
                          include_process=False)
```

Adding custom data sections

You can add a new section to the output by registering a function of your own. Here's an example of how this would be used:

```
def application_data():
    return {"maintainer": "Luis Fernando Gomes",
            "git_repo": "https://github.com/ateliedocodigo/py-healthcheck"
            "config": app.config}
```

```
envdump = EnvironmentDump()
envdump.add_section("application", application_data)
```