sanic-healthcheck Documentation

Release 0.1.1

Erick Daniszewski

Nov 01, 2019

Contents

`sanic-healthcheck` provides a simple way to add health checks and readiness checks to your Sanic application. This makes it easier to monitor your application and ensure it is running in a health state. Monitoring or management tools can ping these endpoints to determine application uptime and status, as well as perform application restart to ensure your application isn't running in a degraded state.

`sanic-healthcheck` was inspired by and borrows from Runscope/healthcheck.

CHAPTER 1

Installing

```
pip install sanic-healthcheck
```

CHAPTER 2

Use Cases

2.1 Docker Compose

Docker Compose allows you to specify health checks in your compose file configuration. With a health check enabled in your application, you can configure your Compose deployment to monitor the health of your application (running on port `3000`):

```
healthcheck:
   test: ['CMD', 'curl', '-f', 'http://localhost:3000/health']
   interval: 10s
   timeout: 3s
   retries: 2
   start_period: 10s
```

2.2 Kubernetes

Kubernetes allows you to define liveness and readiness probes. A health check is effectively equivalent to a liveness check.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: my-application
  name: my-application
spec:
  containers:
  - name: my-application
    image: my/application:1.0
    livenessProbe:
      httpGet:
```

(continues on next page)

(continued from previous page)

```
        path: /health
        port: 3000
      initialDelaySeconds: 10s
      periodSeconds: 10s
    readinessProbe:
      httpGet:
        path: /ready
        port: 3000
```

CHAPTER 3

License

sanic-healthcheck is licensed under the MIT license. See the project's LICENSE file for details.

3.1 Usage

`sanic-healthcheck` provides two types of checkers: a health check and a readiness check.

3.1.1 Check Functions

Check functions take no arguments and return a tuple of `(bool, str)`, where the boolean describes whether or not the check passed, and the string is the message that is output for the check.

```
def check_db_connection():
    ok = db.ping()
    if ok:
        return True, "successfully pinged DB"
    else:
        return False, "failed to ping DB"
```

Exceptions raised in the check are caught and result in the check returning a failure state.

Check functions may also be asynchronous

```
async def check_db_connection():
    ok = await db.ping()
    if ok:
        return True, "successfully pinged DB"
    else:
        return False, "failed to ping DB"
```

3.1.2 Health Check

The `HealthCheck` class lets you register health check functions which get evaluated whenever the health route (`/health` by default) is called. Since the health route may be called frequently by potentially numerous services, the class supports caching health check results for a short period of time.

```
import random

from sanic import Sanic, response
from sanic_healthcheck import HealthCheck

app = Sanic()
health_check = HealthCheck(app)


@app.route('/')
async def test(request):
    return response.json({'hello', 'world'})


# Define checks for the health check.
def check_health_random():
    if random.random() > 0.9:
        return False, 'the random number is > 0.9'
    return True, 'the random number is <= 0.9'


if __name__ == '__main__':
    health_check.add_check(check_health_random)

    app.run(host='0.0.0.0', port=8000)
```

Where a passing health check would look like:

```
$ curl -i localhost:8000/health
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: 5
Content-Length: 2
Content-Type: text/plain; charset=utf-8

OK
```

and a failing health check would look like:

```
$ curl -i localhost:8000/health
HTTP/1.1 500 Internal Server Error
Connection: keep-alive
Keep-Alive: 5
Content-Length: 6
Content-Type: text/plain; charset=utf-8

FAILED
```

3.1.3 Readiness Check

The `HealthCheck` class lets you register health check functions which get evaluated whenever the health route (`/health` by default) is called. Since the health route may be called frequently by potentially numerous services, the class supports caching health check results for a short period of time.

```
import time

from sanic import Sanic, response
from sanic_healthcheck import ReadyCheck

app = Sanic()
ready_check = ReadyCheck(app)

start = time.time()


@app.route('/')
async def test(request):
    return response.json({'hello', 'world'})


# Define checks for the ready check.
def check_ready():
    if time.time() > start + 7:
        return True, 'ready: seven seconds elapsed'
    return False, 'not ready: seven seconds have not elapsed yet'


if __name__ == '__main__':
    ready_check.add_check(check_ready)

    app.run(host='0.0.0.0', port=8000)
```

Where a passing health check would look like:

```
$ curl -i localhost:8000/health
HTTP/1.1 200 OK
Connection: keep-alive
Keep-Alive: 5
Content-Length: 2
Content-Type: text/plain; charset=utf-8

OK
```

and a failing health check would look like:

```
$ curl -i localhost:8000/health
HTTP/1.1 500 Internal Server Error
Connection: keep-alive
Keep-Alive: 5
Content-Length: 6
Content-Type: text/plain; charset=utf-8

FAILED
```

3.2 API Reference

Below is a complete module reference.

3.2.1 sanic_healthcheck

sanic_healthcheck package

Submodules

sanic_healthcheck.checker module

The base class for all implementations of a checker.

class sanic_healthcheck.checker.**BaseChecker**(*app: Optional[sanic.app.Sanic] = None*, *uri: Optional[str] = None*, *checks: Optional[Iterator[Callable]] = None*, *success_handler: Optional[Callable] = None*, *success_headers: Optional[Mapping[KT, VT_co]] = None*, *success_status: Optional[int] = 200*, *failure_handler: Optional[Callable] = None*, *failure_headers: Optional[Mapping[KT, VT_co]] = None*, *failure_status: Optional[int] = 500*, *exception_handler: Optional[Callable] = None*, ***options*)

Bases: `object`

The base class for all checkers.

This class implements various common functionality for all checkers and requires that each checker define its own `run` method. Each checker implementation should also set its own `default_uri`.

Parameters

- **app** – The Sanic application instance to register the checker to. If not specified on initialization, the user must pass it to the `init` method to register the checker route with the application. If specified on initialization, `init` will be called automatically.
- **uri** – The route URI to expose for the checker.
- **checks** – A collection of checks to register with the checker on init. A check is a function which takes no arguments and returns (`bool`, `str`), where the boolean signifies whether the check passed or not, and the string is a message associated with the success/failure.
- **success_handler** – A handler function which takes the check results (a list[dict]) and returns a message string. This is called when all checks pass.
- **success_headers** – Headers to include in the checker response on success. By default, no additional headers are sent. This can be useful if, for example, a success handler is specified which returns a JSON message. The Content-Type: application/json header could be included here.
- **success_status** – The HTTP status code to use when the checker passes its checks.
- **failure_handler** – A handler function which takes the check results (a list[dict]) and returns a message string. This is called when any check fails.

- **failure_headers** – Headers to include in the checker response on failure. By default, no additional headers are sent. This can be useful if, for example, a failure handler is specified which returns a JSON message. The Content-Type: application/json header could be included here.
- **failure_status** – The HTTP status code to use when the checker fails its checks.
- **exception_handler** – A function which would get called when a registered check raises an exception. This handler must take two arguments: the check function which raised the exception, and the tuple returned by `sys.exc_info`. It must return a tuple of (bool, string), where the boolean is whether or not it passed and the string is the message to use for the check response. By default, no exception handler is registered, so an exception will lead to a check failure.
- **options** – Any additional options to pass to the `Sanic.add_route` method on `init`.

add_check(*fn: Callable*) → None
Add a check to the checker.

A check function is a function which takes no arguments and returns (`bool`, `str`), where the boolean signifies whether the check passed or not, and the string is a message associated with the success/failure.

Parameters **fn** – The check to add.

default_uri = None

exec_check(*check: Callable*) → Dict[KT, VT]
Execute a single check and generate a dictionary result from the result of the check.

Parameters **check** – The check function to execute.

Returns A dictionary containing the results of the check.

init(*app: sanic.app.Sanic*, *uri: Optional[str] = None*) → None
Initialize the checker with the Sanic application.

This method will register a new endpoint for the specified Sanic application which exposes the results of the checker.

Parameters

- **app** – The Sanic application to register a new endpoint with.
- **uri** – The URI of the endpoint to register. If not specified, the checker's `default_uri` is used.

run(*request*) → sanic.response.HTTPResponse
Run the checker.

Each subclass of the BaseChecker must define its own `run` logic.

sanic_healthcheck.handlers module

Success and failure handler definitions for checkers.

sanic_healthcheck.handlers.**json_failure_handler**(*results: Iterator[Mapping[KT, VT_co]]*) → str
A failure handler which returns results in a JSON-formatted response.

Parameters **results** –

The results of all checks which were executed for a checker. Each result dictionary is guaranteed to have the keys: 'check', 'message', 'passed', 'timestamp'.

Returns: The checker response, formatted as JSON.

sanic_healthcheck.handlers.**json_success_handler**(*results: Iterator[Mapping[KT, VT_co]]*) → str

A success handler which returns results in a JSON-formatted response.

Parameters **results** – The results of all checks which were executed for a checker. Each result dictionary is guaranteed to have the keys: 'check', 'message', 'passed', 'timestamp'.

Returns The checker response, formatted as JSON.

sanic_healthcheck.health module

A checker for application health.

When configured with a Sanic application, this checker provides a means for the application to specify whether or not it is operating in a healthy state. By identifying broken/unhealthy states, a management system could restart the application, potentially allowing it to recover.

This checker can be used to set up liveness probes for Kubernetes deployments: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-a-liveness-command

It may also be used to define container health checks in docker-compose: https://docs.docker.com/compose/compose-file/#healthcheck

This checker exposes the `/health` endpoint by default.

class sanic_healthcheck.health.**HealthCheck**(*app: Optional[sanic.app.Sanic] = None, uri: Optional[str] = None, checks=None, no_cache: bool = False, success_handler: Optional[Callable] = None, success_headers: Optional[Mapping[KT, VT_co]] = None, success_status: Optional[int] = 200, success_ttl: Optional[int] = 25, failure_handler: Optional[Callable] = None, failure_headers: Optional[Mapping[KT, VT_co]] = None, failure_status: Optional[int] = 500, failure_ttl: Optional[int] = 5, exception_handler: Optional[Callable] = None, **options*)

Bases: *sanic_healthcheck.checker.BaseChecker*

A checker allowing a Sanic application to describe the health of the application at runtime.

The results of registered check functions are cached by this checker by default. To disable result caching, initialize the checker with `no_cache=True`. Since the health endpoint may be polled frequently (and potentially by multiple systems), the cache allows the check function results to be valid for a window of time, reducing the execution cost. This may be particularly helpful if a given health check is more expensive.

Parameters

- **app** – The Sanic application instance to register the checker to. If not specified on initialization, the user must pass it to the `init` method to register the checker route with the application. If specified on initialization, `init` will be called automatically.
- **uri** – The route URI to expose for the checker.
- **checks** – A collection of checks to register with the checker on init. A check is a function which takes no arguments and returns (`bool`, `str`), where the boolean signifies whether the check passed or not, and the string is a message associated with the success/failure.

- **no_cache** – Disable the checker from caching check results. If this is set to `True`, the `success_ttl` and `failure_ttl` do nothing.
- **success_handler** – A handler function which takes the check results (a list[dict]) and returns a message string. This is called when all checks pass.
- **success_headers** – Headers to include in the checker response on success. By default, no additional headers are sent. This can be useful if, for example, a success handler is specified which returns a JSON message. The Content-Type: application/json header could be included here.
- **success_status** – The HTTP status code to use when the checker passes its checks.
- **success_ttl** – The TTL for a successful check result to live in the cache before it is updated.
- **failure_handler** – A handler function which takes the check results (a list[dict]) and returns a message string. This is called when any check fails.
- **failure_headers** – Headers to include in the checker response on failure. By default, no additional headers are sent. This can be useful if, for example, a failure handler is specified which returns a JSON message. The Content-Type: application/json header could be included here.
- **failure_status** – The HTTP status code to use when the checker fails its checks.
- **failure_ttl** – The TTL for a failed check result to live in the cache before it is updated.
- **exception_handler** – A function which would get called when a registered check raises an exception. This handler must take two arguments: the check function which raised the exception, and the tuple returned by `sys.exc_info`. It must return a tuple of (bool, string), where the boolean is whether or not it passed and the string is the message to use for the check response. By default, no exception handler is registered, so an exception will lead to a check failure.
- **options** – Any additional options to pass to the `Sanic.add_route` method on `init`.

default_uri = '/health'

run(*request*) → sanic.response.HTTPResponse
Run all checks and generate an HTTP response for the results.

sanic_healthcheck.ready module

A checker for application readiness.

When configured with a Sanic application, this checker provides a means for the application to specify whether or not the application is in a state where it is fully started up and ready to receive traffic and run normally.

This checker can be used to set up readiness probes for Kubernetes deployments: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-readiness-probes

This checker exposes the `/ready` endpoint by default.

class sanic_healthcheck.ready.**ReadyCheck**(*app: Optional[sanic.app.Sanic] = None*, *uri: Optional[str] = None*, *checks: Optional[Iterator[Callable]] = None*, *success_handler: Optional[Callable] = None*, *success_headers: Optional[Mapping[KT, VT_co]] = None*, *success_status: Optional[int] = 200*, *failure_handler: Optional[Callable] = None*, *failure_headers: Optional[Mapping[KT, VT_co]] = None*, *failure_status: Optional[int] = 500*, *exception_handler: Optional[Callable] = None*, ***options*)

Bases: *sanic_healthcheck.checker.BaseChecker*

A checker allowing a Sanic application to describe when it is ready to serve requests.

The results of registered check functions are not cached by this checker. There should not be a delay in determining application readiness due to a stale cache result.

default_uri = '/ready'

run(*request*) → sanic.response.HTTPResponse

Run all checks and generate an HTTP response for the results.

Module contents

sanic_healthcheck: health checks for your Sanic applications.

class sanic_healthcheck.**HealthCheck**(*app: Optional[sanic.app.Sanic] = None*, *uri: Optional[str] = None*, *checks=None*, *no_cache: bool = False*, *success_handler: Optional[Callable] = None*, *success_headers: Optional[Mapping[KT, VT_co]] = None*, *success_status: Optional[int] = 200*, *success_ttl: Optional[int] = 25*, *failure_handler: Optional[Callable] = None*, *failure_headers: Optional[Mapping[KT, VT_co]] = None*, *failure_status: Optional[int] = 500*, *failure_ttl: Optional[int] = 5*, *exception_handler: Optional[Callable] = None*, ***options*)

Bases: *sanic_healthcheck.checker.BaseChecker*

A checker allowing a Sanic application to describe the health of the application at runtime.

The results of registered check functions are cached by this checker by default. To disable result caching, initialize the checker with `no_cache=True`. Since the health endpoint may be polled frequently (and potentially by multiple systems), the cache allows the check function results to be valid for a window of time, reducing the execution cost. This may be particularly helpful if a given health check is more expensive.

Parameters

- **app** – The Sanic application instance to register the checker to. If not specified on initialization, the user must pass it to the `init` method to register the checker route with the application. If specified on initialization, `init` will be called automatically.
- **uri** – The route URI to expose for the checker.
- **checks** – A collection of checks to register with the checker on init. A check is a function which takes no arguments and returns (`bool`, `str`), where the boolean signifies whether the check passed or not, and the string is a message associated with the success/failure.

- **no_cache** – Disable the checker from caching check results. If this is set to `True`, the `success_ttl` and `failure_ttl` do nothing.
- **success_handler** – A handler function which takes the check results (a list[dict]) and returns a message string. This is called when all checks pass.
- **success_headers** – Headers to include in the checker response on success. By default, no additional headers are sent. This can be useful if, for example, a success handler is specified which returns a JSON message. The Content-Type: application/json header could be included here.
- **success_status** – The HTTP status code to use when the checker passes its checks.
- **success_ttl** – The TTL for a successful check result to live in the cache before it is updated.
- **failure_handler** – A handler function which takes the check results (a list[dict]) and returns a message string. This is called when any check fails.
- **failure_headers** – Headers to include in the checker response on failure. By default, no additional headers are sent. This can be useful if, for example, a failure handler is specified which returns a JSON message. The Content-Type: application/json header could be included here.
- **failure_status** – The HTTP status code to use when the checker fails its checks.
- **failure_ttl** – The TTL for a failed check result to live in the cache before it is updated.
- **exception_handler** – A function which would get called when a registered check raises an exception. This handler must take two arguments: the check function which raised the exception, and the tuple returned by `sys.exc_info`. It must return a tuple of (bool, string), where the boolean is whether or not it passed and the string is the message to use for the check response. By default, no exception handler is registered, so an exception will lead to a check failure.
- **options** – Any additional options to pass to the `Sanic.add_route` method on `init`.

default_uri = '/health'

run(*request*) → sanic.response.HTTPResponse
Run all checks and generate an HTTP response for the results.

class sanic_healthcheck.**ReadyCheck**(*app: Optional[sanic.app.Sanic] = None*, *uri: Optional[str] = None*, *checks: Optional[Iterator[Callable]] = None*, *success_handler: Optional[Callable] = None*, *success_headers: Optional[Mapping[KT, VT_co]] = None*, *success_status: Optional[int] = 200*, *failure_handler: Optional[Callable] = None*, *failure_headers: Optional[Mapping[KT, VT_co]] = None*, *failure_status: Optional[int] = 500*, *exception_handler: Optional[Callable] = None*, ***options*)

Bases: *sanic_healthcheck.checker.BaseChecker*

A checker allowing a Sanic application to describe when it is ready to serve requests.

The results of registered check functions are not cached by this checker. There should not be a delay in determining application readiness due to a stale cache result.

default_uri = '/ready'

run(*request*) → sanic.response.HTTPResponse
Run all checks and generate an HTTP response for the results.

Python Module Index

s

Index