Python

Apr 19, 2021

Contents:

This project checks for various conditions and provides reports when anomalous behavior is detected. Many of these checks involve connecting to back-end services and ensuring basic operations are successful.

CHAPTER 1

django-health-check

This project checks for various conditions and provides reports when anomalous behavior is detected.

The following health checks are bundled with this project:

- cache
- database
- storage
- disk and memory utilization (via `psutil`)
- AWS S3 storage
- Celery task queue
- Celery ping
- RabbitMQ
- Migrations

Writing your own custom health checks is also very quick and easy.

We also like contributions, so don't be afraid to make a pull request.

1.1 Use Cases

The primary intended use case is to monitor conditions via HTTP(S), with responses available in HTML and JSON formats. When you get back a response that includes one or more problems, you can then decide the appropriate course of action, which could include generating notifications and/or automating the replacement of a failing node with a new one. If you are monitoring health in a high-availability environment with a load balancer that returns responses from multiple nodes, please note that certain checks (e.g., disk and memory usage) will return responses specific to the node selected by the load balancer.

1.2 Supported Versions

We officially only support the latest version of Python as well as the latest version of Django and the latest Django LTS version.

1.3 Installation

First install the `django-health-check` package:

```
pip install django-health-check
```

Add the health checker to a URL you want to use:

```
urlpatterns = [
    # ...
    url(r'^ht/', include('health_check.urls')),
]
```

Add the `health_check` applications to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    # ...
    'health_check',                             # required
    'health_check.db',                          # stock Django health checkers
    'health_check.cache',
    'health_check.storage',
    'health_check.contrib.migrations',
    'health_check.contrib.celery',              # requires celery
    'health_check.contrib.celery_ping',         # requires celery
    'health_check.contrib.psutil',              # disk and memory utilization;␣
↪requires psutil
    'health_check.contrib.s3boto3_storage',     # requires boto3 and S3BotoStorage␣
↪backend
    'health_check.contrib.rabbitmq',            # requires RabbitMQ broker
    'health_check.contrib.redis',               # requires Redis broker
]
```

Note : If using `boto 2.x.x` use `health_check.contrib.s3boto_storage`

(Optional) If using the `psutil` app, you can configure disk and memory threshold settings; otherwise below defaults are assumed. If you want to disable one of these checks, set its value to `None`.

```
HEALTH_CHECK = {
    'DISK_USAGE_MAX': 90,  # percent
    'MEMORY_MIN': 100,     # in MB
}
```

If using the DB check, run migrations:

```
django-admin migrate
```

To use the RabbitMQ healthcheck, please make sure that there is a variable named `BROKER_URL` on django.conf.settings with the required format to connect to your rabbit server. For example:

```
BROKER_URL = amqp://myuser:mypassword@localhost:5672/myvhost
```

To use the Redis healthcheck, please make sure that there is a variable named `REDIS_URL` on django.conf.settings with the required format to connect to your redis server. For example:

```
REDIS_URL = redis://localhost:6370
```

1.4 Setting up monitoring

You can use tools like Pingdom or other uptime robots to monitor service status. The `/ht/` endpoint will respond a HTTP 200 if all checks passed and a HTTP 500 if any of the tests failed.

```
$ curl -v -X GET -H http://www.example.com/ht/

> GET /ht/ HTTP/1.1
> Host: www.example.com
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/html; charset=utf-8

<!-- This is an excerpt -->
<div class="container">
    <h1>System status</h1>
    <table>
        <tr>
            <td class="status_1"></td>
            <td>CacheBackend</td>
            <td>working</td>
        </tr>
        <tr>
            <td class="status_1"></td>
            <td>DatabaseBackend</td>
            <td>working</td>
        </tr>
        <tr>
            <td class="status_1"></td>
            <td>S3BotoStorageHealthCheck</td>
            <td>working</td>
        </tr>
    </table>
</div>
```

1.5 Getting machine readable JSON reports

If you want machine readable status reports you can request the `/ht/` endpoint with the `Accept` HTTP header set to `application/json` or pass `format=json` as a query parameter.

The backend will return a JSON response:

```
$ curl -v -X GET -H "Accept: application/json" http://www.example.com/ht/
```

(continues on next page)

(continued from previous page)

```
> GET /ht/ HTTP/1.1
> Host: www.example.com
> Accept: application/json
>
< HTTP/1.1 200 OK
< Content-Type: application/json

{
    "CacheBackend": "working",
    "DatabaseBackend": "working",
    "S3BotoStorageHealthCheck": "working"
}

$ curl -v -X GET http://www.example.com/ht/?format=json

> GET /ht/?format=json HTTP/1.1
> Host: www.example.com
>
< HTTP/1.1 200 OK
< Content-Type: application/json

{
    "CacheBackend": "working",
    "DatabaseBackend": "working",
    "S3BotoStorageHealthCheck": "working"
}
```

1.6 Writing a custom health check

Writing a health check is quick and easy:

```
from health_check.backends import BaseHealthCheckBackend

class MyHealthCheckBackend(BaseHealthCheckBackend):
    #: The status endpoints will respond with a 200 status code
    #: even if the check errors.
    critical_service = False

    def check_status(self):
        # The test code goes here.
        # You can use `self.add_error` or
        # raise a `HealthCheckException`,
        # similar to Django's form validation.
        pass

    def identifier(self):
        return self.__class__.__name__  # Display name on the endpoint.
```

After writing a custom checker, register it in your app configuration:

```
from django.apps import AppConfig

from health_check.plugins import plugin_dir

```

(continues on next page)

(continued from previous page)

```
class MyAppConfig(AppConfig):
    name = 'my_app'

    def ready(self):
        from .backends import MyHealthCheckBackend
        plugin_dir.register(MyHealthCheckBackend)
```

Make sure the application you write the checker into is registered in your `INSTALLED_APPS`.

1.7 Customizing output

You can customize HTML or JSON rendering by inheriting from `MainView` in `health_check.views` and customizing the `template_name`, `get`, `render_to_response` and `render_to_response_json` properties:

```
# views.py
from health_check.views import MainView

class HealthCheckCustomView(MainView):
    template_name = 'myapp/health_check_dashboard.html'  # customize the used␣
↪templates

    def get(self, request, *args, **kwargs):
        plugins = []
        status = 200 # needs to be filled status you need
        # ...
        if 'application/json' in request.META.get('HTTP_ACCEPT', ''):
            return self.render_to_response_json(plugins, status)
        return self.render_to_response(plugins, status)

    def render_to_response(self, plugins, status):       # customize HTML output
        return HttpResponse('COOL' if status == 200 else 'SWEATY', status=status)

    def render_to_response_json(self, plugins, status):  # customize JSON output
        return JsonResponse(
            {str(p.identifier()): 'COOL' if status == 200 else 'SWEATY' for p in␣
↪plugins},
            status=status
        )

# urls.py
import views

urlpatterns = [
    # ...
    url(r'^ht/$', views.HealthCheckCustomView.as_view(), name='health_check_custom'),
]
```

1.8 Django command

You can run the Django command *health_check* to perform your health checks via the command line, or periodically with a cron, as follow:

```
django-admin health_check
```

This should yield the following output:

```
DatabaseHealthCheck      ... working
CustomHealthCheck        ... unavailable: Something went wrong!
```

Similar to the http version, a critical error will cause the command to quit with the exit code *1*.

1.9 Other resources

- django-watchman is a package that does some of the same things in a slightly different way.
- See this weblog about configuring Django and health checking with AWS Elastic Load Balancer.

CHAPTER 2

contrib

2.1 psutil

Full disks and out-of-memory conditions are common causes of service outages. These situations can be averted by checking disk and memory utilization via the `psutil` package:

```
pip install psutil
```

Once that dependency has been installed, make sure that the corresponding Django app has been added to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    # ...
    'health_check',                             # required
    'health_check.contrib.psutil',              # disk and memory utilization;␣
↪requires psutil
    # ...
]
```

The following default settings will be used to check for disk and memory utilization. If you would prefer different thresholds, you can add the dictionary below to your Django settings file and adjust the values accordingly. If you want to disable any of these checks, set its value to `None`.

```
HEALTH_CHECK = {
    'DISK_USAGE_MAX': 90,   # percent
    'MEMORY_MIN' = 100,     # in MB
}
```

2.2 celery

If you are using Celery you may choose between two different Celery checks.

health_check.contrib.celery sends a task to the queue and it expects that task to be executed in *HEALTHCHECK_CELERY_TIMEOUT* seconds which by default is three seconds. You may override that in your Django settings module. This check is suitable for use cases which require that tasks can be processed frequently all the time.

health_check.contrib.celery_ping is a different check. It checks that each predefined Celery task queue has a consumer (i.e. worker) that responds *{"ok": "pong"}* in *HEALTHCHECK_CELERY_PING_TIMEOUT* seconds. The default for this is one second. You may override that in your Django settings module. This check is suitable for use cases which don't require that tasks are executed almost instantly but require that they are going to be executed in sometime the future i.e. that the worker process is alive and processing tasks all the time.

You may also use both of them. To use these checks add them to *INSTALLED_APPS* in your Django settings module.

CHAPTER 3

Settings

Settings can be configured via the `HEALTH_CHECK` dictionary.

`WARNINGS_AS_ERRORS`
: Treats `ServiceWarning` as errors, meaning they will cause the views to respond with a 500 status code. Default is `True`. If set to `False` warnings will be displayed in the template on in the JSON response but the status code will remain a 200.

3.1 Security

Django health check can be used as a possible DOS attack vector as it can put your system under a lot of stress. As a default the view is also not cached by CDNs. Therefore we recommend to use a secure token to protect you application servers from an attacker.

1. Setup HTTPS. Seriously...
2. Add a secure token to your URL.

Create a secure token:

```
python -c "import secrets; print(secrets.token_urlsafe())"
```

Add it to your URL:

```
urlpatterns = [
    # ...
    url(r'^ht/super_secret_token/'), include('health_check.urls')),
]
```

You can still use any uptime bot that is URL based while enjoying token protection.

Warning: Do NOT use Django's *SECRET_KEY* setting. This should never be exposed, to any third party. Not even your trusted uptime bot.

3.2 psutil

The following default settings will be used to check for disk and memory utilization. If you would prefer different thresholds, you can add the dictionary below to your Django settings file and adjust the values accordingly. If you want to disable any of these checks, set its value to `None`.

```
HEALTH_CHECK = {
    'DISK_USAGE_MAX': 90,   # percent
    'MEMORY_MIN' = 100,     # in MB
}
```

With the above default settings, warnings will be reported when disk utilization exceeds 90% or available memory drops below 100 MB.

DISK_USAGE_MAX
: Specify the desired disk utilization threshold, in percent. When disk usage exceeds the specified value, a warning will be reported.

MEMORY_MIN
: Specify the desired memory utilization threshold, in megabytes. When available memory falls below the specified value, a warning will be reported.

3.3 Celery Health Check

Using *django.settings* you may exert more fine-grained control over the behavior of the celery health check

Table 1: Additional Settings

Name	Type	Default	Description
HEALTHCHECK_CELERY_QUEUE_TIMEOUT	Number	3	Specifies the maximum amount of time a task may spend in the queue before being automatically revoked with a *TaskRevokedError*.
HEALTHCHECK_CELERY_RESULT_TIMEOUT	Number	3	Specifies the maximum total time for a task to complete and return a result, including queue time.

CHAPTER 4

ChangeLog

This package is released on GitHub. Please refer to the GitHub release page to review the changes in each version.

https://github.com/KristianOellegaard/django-health-check/releases

Index