GeoHealthCheck Documentation

Release 0.5.0

Tom Kralidis

Nov 15, 2018

Contents

CHAPTER 1

Overview

GeoHealthCheck (GHC) is a Python application to support monitoring OGC services uptime, availability and Quality of Service (QoS).

It can be used to monitor overall health of OGC services like WMS, WFS, WCS, WMTS, SOS, CSW and more, but also standard web(-API) URLs.

CHAPTER 2

Features

- lightweight (Python with Flask)
- easy setup
- support for numerous OGC resources
- flexible and customizable: look and feel, scoring matrix
- user management
- database agnostic: any SQLAlchemy supported backend
- database upgrades: using Alembic with Flask-Migrate
- extensible healthchecks via Plugins
- per-resource scheduling and notifications

CHAPTER 3

Links

- website: http://geohealthcheck.org
- GitHub: https://github.com/geopython/geohealthcheck
- Demo: https://demo.geohealthcheck.org (official demo, master branch)
- Presentation: http://geohealthcheck.org/presentation
- Gitter Chat: https://gitter.im/geopython/GeoHealthCheck

This document applies to GHC version 0.5.0 and was generated on Nov 15, 2018. The latest version is always available at http://docs.geohealthcheck.org.

Contents:

3.1 Installation

Below are installation notes for GeoHealthCheck (GHC).

3.1.1 Docker

The easiest and quickest install for GHC is with Docker/Docker Compose using the GHC images hosted on Docker Hub.

See the GHC Docker Readme for a full guide.

3.1.2 Requirements

GeoHealthCheck is built on the awesome Flask micro-framework and uses *Flask-SQLAlchemy* for database interaction and Flask-Login for authorization. *Flask-Migrate* with *Alembic* and *Flask-Script* is used for database upgrades.

OWSLib is used to interact with OGC Web Services.

APScheduler is used to run scheduled healthchecks.

These dependencies are automatically installed (see below). `Paver` is used for installation and management. `Cron` was used for scheduling the actual healthchecks before v0.5.0.

3.1.3 Install

Note: It is strongly recommended to install GeoHealthCheck in a Python `virtualenv`. a `virtualenv` is self-contained and provides the flexibility to install / tear down / whatever packages without affecting system wide packages or settings. If installing on Ubuntu, you may need to install the python-dev package for installation to complete successfully.

- Download a GeoHealthCheck release from https://github.com/geopython/GeoHealthCheck/releases, or clone manually from GitHub.

```
virtualenv ghc && cd ghc
. bin/activate
git clone https://github.com/geopython/GeoHealthCheck.git
cd GeoHealthCheck

# install paver dependency for admin tool
pip install Paver

# setup app
paver setup

# create secret key to use for auth
paver create_secret_key

# almost there!  Customize config
vi instance/config_site.py
# edit:
# - SQLALCHEMY_DATABASE_URI
# - SECRET_KEY  # from paver create_secret_key
# - GHC_RETENTION_DAYS
# - GHC_SELF_REGISTER
# - GHC_NOTIFICATIONS
# - GHC_NOTIFICATIONS_VERBOSITY
# - GHC_ADMIN_EMAIL
# - GHC_NOTIFICATIONS_EMAIL
# - GHC_SITE_TITLE
# - GHC_SITE_URL
# - GHC_RUNNER_IN_WEBAPP # see 'running' section below
# - GHC_SMTP  # if GHC_NOTIFICATIONS is enabled
# - GHC_MAP  # or use default settings

# init database
python GeoHealthCheck/models.py create

# start web-app
python GeoHealthCheck/app.py  # http://localhost:8000/
```

NB GHC supports internal scheduling, no cronjobs required.

3.1.4 Upgrade

An existing GHC database installation can be upgraded with:

```
# In the top directory (e.g. the topdir cloned from github)
paver upgrade

# Notice any output, in particular errors
```

Notes:

- **Always backup your database first!!**
- make sure Flask-Migrate is installed (see requirements.txt), else: *pip install Flask-Migrate==2.0.3*, but best is to run *paver setup* also for other dependencies
- upgrading is "smart": you can always run *paver upgrade*, it has no effect when DB is already up to date
- when upgrading from earlier versions without Plugin-support:
  - adapt your *config_site.py* to Plugin settings from *config_main.py*
  - assign *Probes* and *Checks* to each *Resource* via the UI

When running with Docker see the GHC Docker Readme how to run *paver upgrade* within your Docker Container.

Upgrade notes v0.5.0

In GHC v0.5.0 a new run-architecture was introduced. By default, healthchecks run under the control of an internal scheduler, i.s.o. of external cron-jobs. See also the *Architecture* chapter and *Running Healthchecks* and below.

3.1.5 Running

Start using Flask's built-in WSGI server:

```
python GeoHealthCheck/app.py  # http://localhost:8000
python GeoHealthCheck/app.py 0.0.0.0:8881  # http://localhost:8881
python GeoHealthCheck/app.py 192.168.0.105:8957  # http://192.168.0.105:8957
```

This runs the (Flask) **GHC Webapp**, by default with the **GHC Runner** (scheduled healthchecker) internally. See also *Running Healthchecks* for the different options running the **GHC Webapp** and **GHC Runner**. It is recommended to run these as separate processes. For this set **GHC_RUNNER_IN_WEBAPP** to *False* in your *site_config.py*. From the command-line run both processes, e.g. in background or different terminal sessions:

```
# run GHC Runner, here in background
python GeoHealthCheck/scheduler.py &

# run GHC Webapp for http://localhost:8000
python GeoHealthCheck/app.py
```

To enable in Apache, use `GeoHealthCheck.wsgi` and configure in Apache as per the main Flask documentation.

3.1.6 Running under a sub-path

By default GeoHealthCheck is configured to run under the root directory on the webserver. However, it can be configured to run under a sub-path. The method for doing this depends on the webserver you are using, but the general requirement is to pass Flask's `SCRIPT_NAME` environment variable when GeoHealthCheck is started.

Below is an example of how to use nginx and gunicorn to run GeoHealthCheck in a directory "geohealthcheck", assuming that you have nginx and gunicorn already set up and configured:

- In nginx add a section to the server block you are running GeoHealthCheck under:

```
location /geohealthcheck {
  proxy_pass http://127.0.0.1:8000/geohealthcheck;
}
```

- Include the parameter "-e SCRIPT_NAME=/geohealthcheck" in your command for running gunicorn:

```
gunicorn -e SCRIPT_NAME=/geohealthcheck app:app
```

3.1.7 Production Recommendations

Use Docker!

When running GHC in long-term production environment the following is recommended:

- use Docker, see the GHC Docker Readme

Using Docker, especially with Docker Compose (sample files provided) is our #1 recommendation. It saves all the hassle from installing the requirements, upgrades etc. Docker (Compose) is also used to run the GHC demo site and almost all of our other deployments.

Use PostgreSQL

Although GHC will work with *SQLite*, this is not a good option for production use, in particular for reliability starting with GHC v0.5.0:

- reliability: **GHC Runner** will do concurrent updates to the database, this will be unreliable under *SQLite*
- performance: PostgreSQL has been proven superior, especially in query-performance

Use a WSGI Server

Although GHC can be run from the commandline using the Flask internal WSGI web-server, this is a fragile and possibly insecure option in production use (as also the Flask manual states). Best is to use a WSGI-server as stated in the Flask deployment options.

See for example the GHC Docker run.sh script to run the GHC Webapp with *gunicorn* and the GHC Runner run-runner.sh script to run the scheduled healthchecks.

Use virtualenv

This is a general Python-recommendation. Save yourself from classpath and library hells by using *virtualenv*!

Use SSL (HTTPS)

As users and admin may login, running on plain http will send passwords in the clear. These days it has become almost trivial to automatically install SSL certificates with Let's Encrypt.

3.2 Configuration

Core configuration is set by GeoHealthCheck in `GeoHealthCheck/config_main.py`. You can override these settings in `instance/config_site.py`:

- **SQLALCHEMY_DATABASE_URI**: the database configuration. See the SQLAlchemy documentation for more info
- **SECRET_KEY**: secret key to set when enabling authentication. Use the output of `paver create_secret_key` to set this value
- **GHC_RETENTION_DAYS**: the number of days to keep run history
- **GHC_PROBE_HTTP_TIMEOUT_SECS**: stop waiting for the first byte of a probe response after the given number of seconds
- **GHC_MINIMAL_RUN_FREQUENCY_MINS**: minimal run frequency for Resource that can be set in web UI
- **GHC_SELF_REGISTER**: allow registrations from users on the website
- **GHC_NOTIFICATIONS**: turn on email notifications
- **GHC_NOTIFICATIONS_VERBOSITY**: receive additional email notifications than just `Failing` and `Fixed` (default `True`)
- **GHC_WWW_LINK_EXCEPTION_CHECK**: turn on checking for OGC Exceptions in `WWW:LINK` Resource responses (default `False`)
- **GHC_ADMIN_EMAIL**: email address of administrator / contact- notification emails will come from this address
- **GHC_NOTIFICATIONS_EMAIL**: list of email addresses that notifications should come to. Use a different address to **GHC_ADMIN_EMAIL** if you have trouble receiving notification emails. Also, you can set separate notification emails to specific resources. Failing resource will send notification to emails from **GHC_NOTIFICATIONS_EMAIL** value and emails configured for that specific resource altogether.
- **GHC_SITE_TITLE**: title used for installation / deployment
- **GHC_SITE_URL**: url of the installation / deployment
- **GHC_SMTP**: configure SMTP settings if **GHC_NOTIFICATIONS** is enabled
- **GHC_RELIABILITY_MATRIX**: classification scheme for grading resource
- **GHC_PLUGINS**: list of Core/built-in Plugin classes or modules available on installation
- **GHC_USER_PLUGINS**: list of Plugin classes or modules provided by user (you)
- **GHC_PROBE_DEFAULTS**: Default *Probe* class to assign on "add" per Resource-type
- **GHC_METADATA_CACHE_SECS**: metadata, "Capabilities Docs", cache expiry time, default 900 secs, -1 to disable
- **GHC_RUNNER_IN_WEBAPP**: should the GHC Runner Daemon be run in webapp (default: `True`)
- **GHC_LOG_LEVEL**: logging level: 10=DEBUG 20=INFO 30=WARN(ING) 40=ERROR 50=FATAL/CRITICAL (default: 30, WARNING)
- **GHC_MAP**: default map settings
  - **url**: URL of TileLayer
  - **centre_lat**: Centre latitude for homepage map
  - **centre_long**: Centre longitude for homepage map

- **maxzoom**: maximum zoom level
- **subdomains**: available subdomains to help with parallel requests

3.2.1 Enabling or disabling languages

Open the file `GeoHealthCheck/app.py` and look for the language switcher (e.g. 'en','fr') and remove or add the desired languages. In case a new language (e.g. this needs a new translation file called `*.po`) is to be added, make a copy of one of the folders in `GeoHealthCheck/translations/`; rename the folder to the desired language (e.g. 'de' for german); start editing the file in `LC_MESSAGES/messages.po` and add your translations to the ''msgstr''. Don't forget the change the specified language in the messages.po file as well. For example the `messages.po` file for the german case has an english ''msgid" string, which needs to be translated in ''msgstr" as seen below.

```
-#: GeoHealthCheck/app.py:394
-msgid "This site is not configured for self-registration"
-msgstr "Diese Webseite unterstützt keine Selbstregistrierung"
```

3.2.2 Customizing the Score Matrix

GeoHealthCheck uses a simple matrix to provide an indication of overall health and / or reliability of a resource. This matrix drives the CSS which displays a given resource's state with a colour. The default matrix is defined as follows:

low	high	score/colour
0	49	red
50	79	orange
80	100	green

To adjust this matrix, edit **GHC_RELIABILITY_MATRIX** in `instance/config_site.py`.

3.2.3 Configuring notifications

GeoHealthCheck can send notifications to various channels, depending on resource. Notifications can be configured in edit form:

There are two channels implemented:

Email

Notifications can be send to designated emails. If set in config, GeoHealthCheck will send notifications for all resources to emails defined in **GHC_NOTIFICATIONS_EMAIL**. Additionally, each resource can have arbitrary list of emails (filled in **Notify emails** field in edit form). By default, when resource is created, owner's email is added to the list. User can add any email address, even for users that are not registered in GeoHealthCheck instance. When editing emails list for a resource, user will get address suggestions based on emails added for other resources by that user. Multiple emails should be separated with comma (,) char.

Webhook

Notifications can be also send as webhooks through *POST* request. Resource can have arbitrary number of webhooks configured.

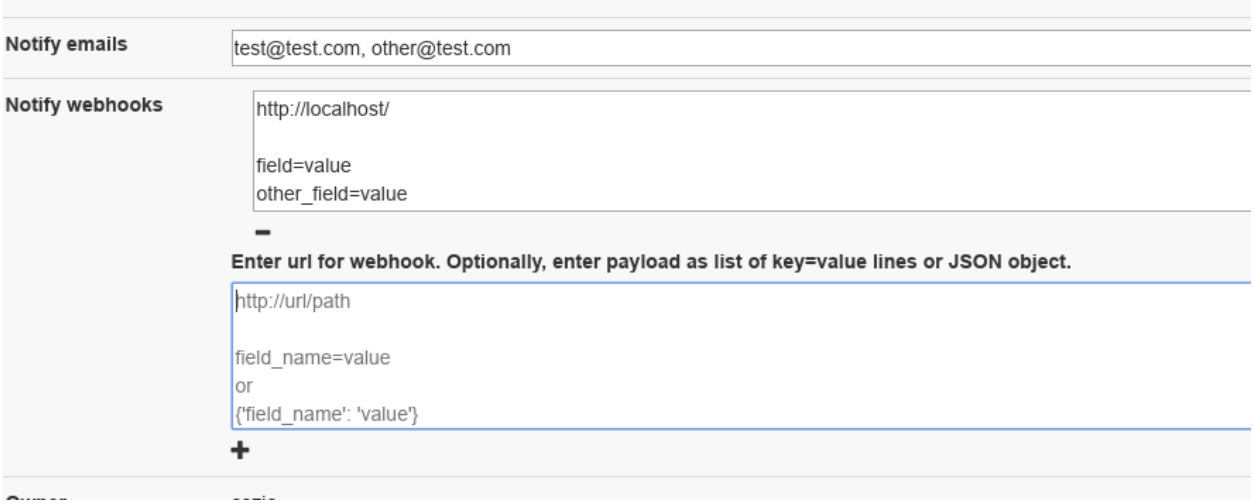

Fig. 1: *Figure - GHC notifications configuration*

In edit form, user can add webhook configuration. Each webhook should be entered in separate field. Each webhook should contain at least URL to which *POST* request will be send. GeoHealthCheck will send following fields with that request:

Form field	Field type	Description
ghc.result	string	Descriptive result of failed test
ghc.resource.url	URL	Resource's url
ghc.resource.title	string	Resource's title
ghc.resource.type	string	Resource's type name
ghc.resource.view	URL	URL to resource data in GeoHealthCheck

Configuration can hold additional form payload that will be send along with GHC fields. Syntax for configuration:

- first line should be url to which webhook will be send
- second line should be empty
- third line (and subsequent) are used to store custom payload, and should contain either: * each pair of field and value in separate lines (*field=value*) * JSONified object, which properties will be used as form fields

Configuration samples:

- just an url

```
http://server/webhook/endpoint
```

- url with fields as field-value pairs

```
http://server/webhook/endpoint

foo=bar
otherfield=someothervalue
```

- url and payload as JSON:

```
http://server/webhook/endpoint

{"foo":"bar","otherfield":"someothervalue"}
```

3.3 Adminstration

3.3.1 Database

For database administration the following commands are available.

NB, although SQLite works fine for most installations it is recommended for performance and reliability to use PostgreSQL.

create db

To create the database execute the following:

Open a command line, (if needed activate your virtualenv), and do

```
python GeoHealthCheck/models.py create
```

drop db

To delete the database execute the following, however you will loose all your information. So please ensure backup if needed:

Open a command line, (if needed activate your virtualenv), and do

```
python GeoHealthCheck/models.py drop
```

Note: you need to create a Database again before you can start GHC again.

load data

To load a JSON data file, do (WARN: deletes existing data!)

```
python GeoHealthCheck/models.py load <datafile.json> [y/n]
```

Hint: see *tests/data* for example JSON data files.

export data

Exporting database-data to a .json file with or without Runs is still to be done.

Exporting Resource and Run data from a running GHC instance can be effected via a REST API, for example:

- all Resources: https://demo.geohealthcheck.org/json (or as CSV)
- one Resource: https://demo.geohealthcheck.org/resource/1/json (or CSV)
- all history (Runs) of one Resource: https://demo.geohealthcheck.org/resource/1/history/json (or in csv)

NB for detailed reporting data only JSON is supported.

3.3.2 User Management

On setup a single *admin* user is created interactively.

Via the **GHC_SELF_REGISTER** config setting, you allow/disallow registrations from users on the website.

3.3.3 Adding Resources

When being logged in, click the Add+ button for adding new resources.

The folowing resource types are available:

- File Transfer Protocol (FTP)
- Web Map Service (WMS)
- Web Address (URL)
- Catalogue Service (CSW)
- Web Map Tile Service (WMTS)
- Web Processing Service (WPS)
- Web Coverage Service (WCS)
- Web Feature Service (WFS)
- Tile Map Service (TMS)
- Web Accessible Folder (WAF)
- Sensor Observation Service (SOS)
- SensorThings API (STA)
- GeoNode autodiscovery

3.3.4 Deleting Resources

Open the resource details by clicking its name in the resources list at the Dashboard page. Under the resource title is a red Delete button.

3.3.5 Editing Resources

Open the resource details by clicking its name in the resources list at the Dashboard page. Under the resource title is a blue Edit button.

The following aspects of a *Resource* can be edited:

- Resource name
- Resource Tags
- Resource active/non-active
- Notification recipients
- Resource run schedule
- Resource Probes, select from “Probes Available”
- For each Probe: Probe parameters

- For each Probe: Probe Checks, select from "Checks Available"
- For each Check: Checks parameters

By default, when resource is created, owner's email will be added to notifications, however, resource can have arbitrary number or emails to notify.

3.3.6 Running Healthchecks

Healthchecks (Runs) for each Resource can be scheduled via *cron* or (starting with v0.5.0) by running the **GHC Runner** app standalone (as daemon) or within the **GHC Webapp**.

Scheduling via Cron

Applies only to pre-0.5.0 versions.

Edit the file `jobs.cron` so that the paths reflect the path to the virtualenv. Set the first argument to the desired monitoring time step. If finished editing, copy the command line calls e.g. `/YOURvirtualenv/bin_or_SCRIPTSonwindows/python /path/to/GeoHealthCheck/GeoHealthCheck/healthcheck.py run` to the commandline to test if they work sucessfully. On Windows - do not forget to include the ''.exe" file extension to the python executable. For documentation how to create cron jobs see your operating system: on *NIX systems e.g. `crontab -e` and on windows e.g. the nssm.

NB the limitation of cron is that the per *Resource* schedule cannot be applied as the cron job will run healthchecks on all *Resources*.

GHC Runner as Daemon

In this mode GHC applies internal scheduling for each individual *Resource*. This is the preferred mode as each *Resource* can have its own schedule (configurable via Dashboard) and *cron* has dependencies on local environment. Later versions may phase out cron-scheduling completely.

The **GHC Runner** can be run via the command *paver runner_daemon* or can run internally within the **GHC Webapp** by setting the config variable **GHC_RUNNER_IN_WEBAPP** to *True* (the default). NB it is still possible to run GHC as in the pre-v0.5.0 mode using cron-jobs: just run the **GHC Webapp** with **GHC_RUNNER_IN_WEBAPP** set to *False* and have your cron-jobs scheduled.

In summary there are three options to run GHC and its healthchecks:

- run **GHC Runner** within the **GHC Webapp**: set **GHC_RUNNER_IN_WEBAPP** to *True* and run only the GHC webapp
- (recommended): run **GHC Webapp** and **GHC Runner** separately (set **GHC_RUNNER_IN_WEBAPP** to *False*)
- (deprecated): run **GHC Webapp** with **GHC_RUNNER_IN_WEBAPP** set to *False* and schedule healthchecks via external cron-jobs

3.3.7 Build Documentation

Open a command line, (if needed activate your virtualenv) and move into the directory `GeoHealthCheck/doc/`. In there, type ''make html" plus ENTER and the documentation should be built locally.

3.3.8 GeoNode Resource Type Notes

GeoNode Resource is a virtual resource. It represents one GeoNode instance, but underneath auto-discovery is applied of OWS endpoints available in that instance. Note, that OWS auto-discovery feature is optional, and you should check if your GeoNode instance has this feature enabled.

When adding *GeoNode instance* Resource, you have to enter the url to the GN instance's home page. GeoHealthCheck will construct the urls to target OWS endpoints listing and create relevant Resources (WMS, WFS, WMTS, OWC Resources). It will check all endpoints provided by the GeoNode API, and will reject those which responded with an error.

All resources added in this way will have at least one tag, which is constructed with template: *GeoNode _hostname_*, where *_hostname_* is a host name from url provided. For example, let's assume you add GeoNode instance that is served from *demo.geonode.org*. All resources created in this way will have *GeoNode demo.geonode.org* tag.

3.4 Architecture

GeoHealthCheck (GHC) consists of three cooperating parts as depicted in the figure below.

The **GHC Webapp** provides the Dashboard where users configure web services (Resources) for (scheduled) health-checks and view the status of these checks. The **GHC Runner** performs the actual health-checks and notifications, based on what the user configured via the GHC Webapp.

The third part is the **Database** that stores all information like users, resources, checks, schedules, results etc.

The **GHC Webapp** is run as a standard Python (Flask) webapp. The **GHC Runner** runs as a daemon process using an internal scheduler to invoke the actual healthchecks.

GHC Webapp and **GHC Runner** can run as separate processes (preferred) or both within the **GHC Webapp** process. This depends on a configuration option. If **GHC_RUNNER_IN_WEBAPP** is set to True (the default) then the **GHC Runner** is started within the **GHC Webapp**.

A third option is to only run the **GHC Webapp** and have the **GHC Runner** scheduled via *cron*. This was the (only) GHC option before v0.5.0 and will be phased out as starting with v0.5.0, per-Resource scheduling was introduced and *cron* support is highly platform-dependent (e.g. hard to use with Docker-based technologies).

Dependent on the database-type (Postgres or SQLite) the **Database** is run within the above processes (SQLite) or as a separate process (Postgres).

So in the most minimal setup, i.e. **GHC Webapp** and **GHC Runner** running within a single process and using SQLite as the database, only a single GHC process instance is required.

3.4.1 Core Concepts

GeoHealthCheck is built with the following concepts in mind:

- *Resource*: a single, unique endpoint, like an OGC WMS, FTP URL, or plain old web link. A GeoHealthCheck deployment typically monitors numerous *Resources*.
- *Run*: the execution and scoring of a test against a *Resource*. A *Resource* may have multiple *Runs*
- Each *User* owns one or more *Resources*
- Each *Resource* is tested, "probed", via one or more *Probes*
- Each *Probe* typically runs one or more requests on a *Resource* URL
- Each *Probe* invokes one or more *Checks* to determine *Run* result

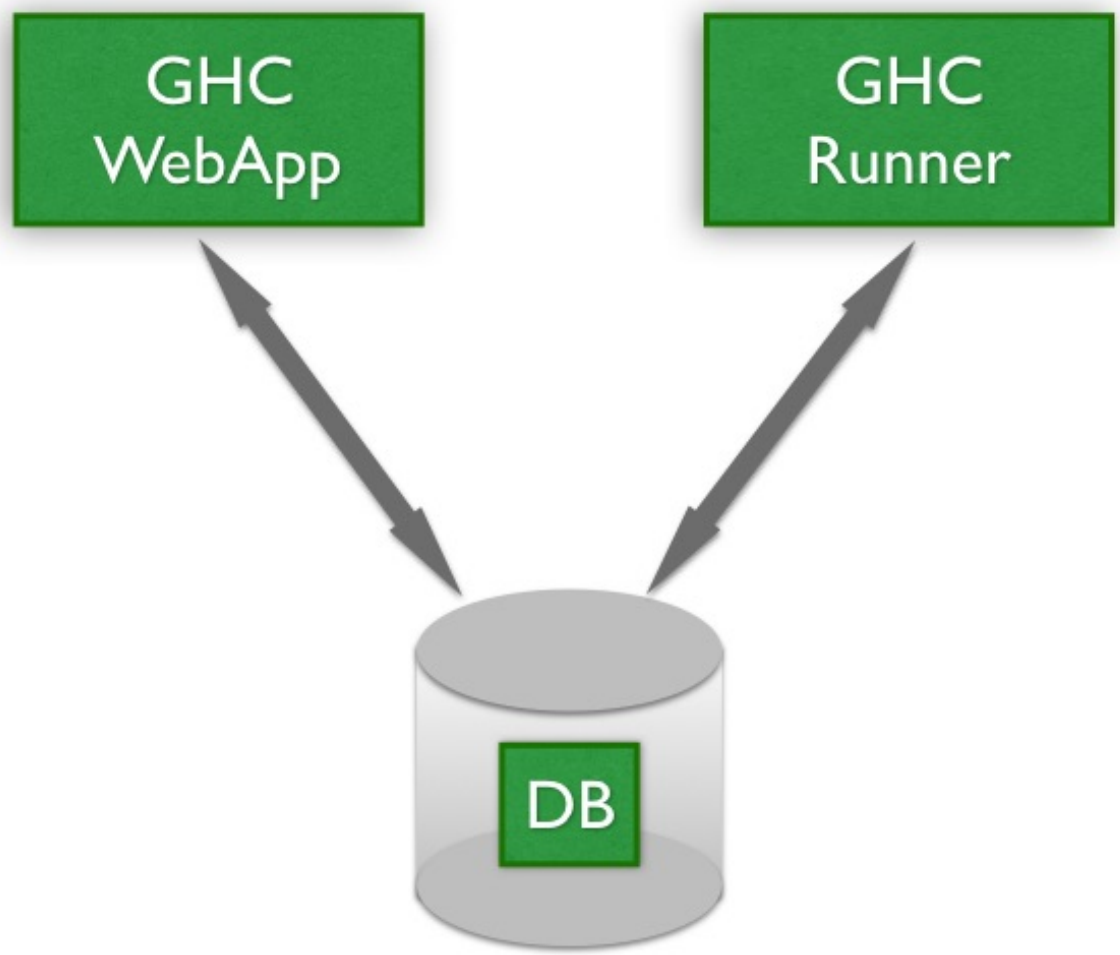

Fig. 2: *Figure - GHC Parts*

- *Probes* and *Checks* are extensible *Plugins* via respective *Probe* and *Check* classes
- One or more *Tags* can be associated with a *Resource* to support grouping
- One or more *Recipient* can be associated with a *Resource*. Each *Recipient* describes:
  - communication channel
  - target identifier

3.4.2 Data Model

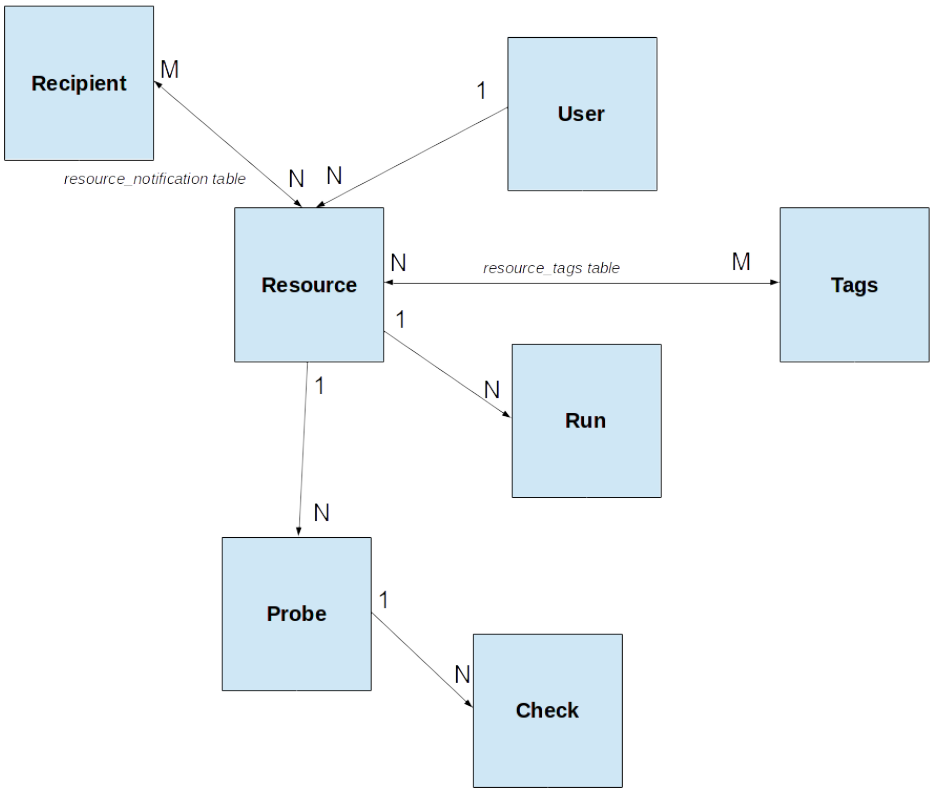

Fig. 3: *Figure - GHC Data Model*

3.4.3 GHC Webapp Design

The **GHC Webapp** is realized as a standard *Flask* web-application using *SQLAlchemy* for ORM database support. It is the user-visible part of GHC as it runs via the browser. Its main two functionalities are to allow users to:

- manage (create, update, delete) Resources, their attributes and their Probes and Checks, and
- view results and statistics of Resources (Dashboard function)

Deployment can be realized using the various standard Flask deployment methods: standalone, within a WSGI server etc.

As an option (via configuration, see above) the **GHC Runner** may run within the **GHC Webapp**. Note that in case that when the **GHC Webapp** runs as multiple processes and/or threads "Resource Locking" (see below) will prevent inconsistencies.

3.4.4 GHC Runner Design

The **GHC Runner** in its core is a job scheduler based on the Python library APScheduler. Each job scheduled is a healthcheck runner for a single *Resource* that runs all the *Probes* for that *Resource*. The run-frequency follows the per-Resource run frequency (since v0.5.0).

The GHC Runner is thus responsible for running the *Probes* for each *Resource*, storing the *Results* and doing notifications when needed.

The **GHC Runner** can run as a separate (Python) process, or within the **GHC WebApp** (see above). Separate processes is the preferred mode of running.

Job Runner Synchronization

As multiple instances of the job scheduler (i.e. APScheduler) may run in different processes and even threads within processes, the database is used to synchronize and assure only one job will run for a single *Resource*.

This is achieved by having one lock per *Resource* via the table *ResourceLock*. Only the process/thread that acquires its related *ResourceLock* record runs the job. As to avoid permanent "lockouts", each *ResourceLock* has a lifetime, namely the timespan until the next Run as configured for/per *Resource*. This gives all job runners a chance to obtain a lock once "time's up" for the *ResourceLock*.

Additional lock-liveliness is realized by using a unique **UUID per job runner**. Once the lock is obtained, the UUID-field of the lock record is set and committed to the DB. If we then try to obtain the lock again (by reading from DB) but the UUID is different this means another job runner instance did the same but was just before us. The lock-lifetime (see above) guards that a particular UUID keeps the lock forever, e.g. on sudden application shutdown.

To further increase liveliness, mainly to avoid all Jobs running at the same time when scheduled to run at the same frequency, each Job is started with a random time-offset on GHC Runner startup.

The locking mechanism described above is supported for SQLite, but it is strongly advised to use PostgreSQL in production deployments, also for better robustness and performance in general.

3.5 Plugins

GHC can be extended for Resource-specific healthchecks via Plugins. GHC already comes with a set of standard plugins that may suffice most installations. However, there is no limit to detailed healthchecks one may want to perform. Hence developers can extend or even replace the GHC standard Plugins with custom implementations.

Two Plugin types exist that can be extended: the *Probe* and *Check* class.

3.5.1 Concepts

GHC versions after May 1, 2017 perform healthchecks exclusively via Plugins (see *Upgrade* how to upgrade from older versions). The basic concept is simple: each *Resource* (typically an OWS endpoint) has one or more *Probes*. During a GHC run (via *cron* or manually), GHC sequentually invokes the *Probes* for each *Resource* to determine the health (QoS) of the *Resource*.

A *Probe* typically implements a single request like a *WMS GetMap*. A *Probe* contains and applies one or more *Checks* (the other Plugin class). A *Check* implements typically a single check on the HTTP Response object of its parent *Probe*,

for example if the HTTP response has no errors or if a *WMS GetMap* actually returns an image (content-type check). Each *Check* will supply a *CheckResult* to its parent *Probe*. The list of *CheckResults* will then ultimately determine the *ProbeResult*. The *Probe* will in turn supply the *ProbeResult* to its parent *ResourceResult*. The GHC healthchecker will then determine the final outcome of the *Run* (fail/success) for the *Resource*, adding the list of Probe/CheckResults to the historic Run-data in the DB. This data can later be used for reporting and determining which *Check(s)* were failing.

So in summary: a *Resource* has one or more *Probes*, each *Probe* one or more *Checks*. On a GHC run these together provide a *Result*.

Probes and Checks available to the GHC instance are configured in *config_site.py*, the GHC instance config file. Also configured there is the default *Probe* class to assign to a Resource-type when it is added. Assignment and configuration/parameterization of *Probes* and *Checks* is via de UI on the Resource-edit page and stored in the database (tables: *probe_vars* and *check_vars*). That way the GHC healthcheck runner can read (from the DB) the list of Probes/Checks and their config for each Resource.

3.5.2 Implementation

Probes and *Checks* plugins are implemented as Python classes derived from `GeoHealthCheck.probe.Probe` and `GeoHealthCheck.check.Check` respectively. These classes inherit from the GHC abstract base class `GeoHealthCheck.plugin.Plugin`. This class mainly provides default attributes (in capitals) and introspection methods needed for UI configuration. *Class-attributes* (in capitals) are the most important concept of GHC Plugins in general. These provide metadata for various GHC functions (internal, UI etc). General class-attributes that Plugin authors should provide for derived *Probes* or *Checks* are:

- *AUTHOR*: Plugin author or team.
- *NAME*: Short name of Plugin.
- *DESCRIPTION*: Longer description of Plugin.
- *PARAM_DEFS*: Plugin Parameter definitions (see next)

PARAM_DEFS, a Python *dict* defines the parameter definitions for the *Probe* or *Check* that a user can configure via the UI. Each parameter (name) is itself a *dict* entry key that with the following key/value pairs:

- *type*: the parameter type, value: 'string', 'stringlist' (comma-separated strings) or 'bbox' (lowerX, lowerY, upperX, upperY),
- *description*: description of the parameter,
- *default*: parameter default value,
- *required*: is parameter required?,
- *range*: range of possible parameter values (array of strings), results in UI dropdown selector

A *Probe* should supply these additional class-attributes:

- *RESOURCE_TYPE* : GHC Resource type this Probe applies to, e.g. *OGC:WMS*, **:** (any Resource Type), see *enums.py* for range
- *REQUEST_METHOD* : HTTP request method capitalized, 'GET' (default) or 'POST'.
- *REQUEST_HEADERS* : *dict* of optional HTTP request headers
- *REQUEST_TEMPLATE*: template in standard Python *str.format(*args)* to be substituted with actual parameters from *PARAM_DEFS*
- *CHECKS_AVAIL* : available Check (classes) for this Probe.

Note: *CHECKS_AVAIL* denotes all possible *Checks* that can be assigned, by default or via UI, to an instance of this *Probe*.

A *Check* has no additional class-attributes.

In many cases writing a *Probe* is a matter of just defining the above class-attributes. The GHC healthchecker `GeoHealthCheck.healthcheck.run_test_resource()` will call lifecycle methods of the `GeoHealthCheck.probe.Probe` base class, using the class-attributes and actualized parameters (stored in *probe_vars* table) as defined in *PARAM_DEFS* plus a list of the actual and parameterized Checks (stored in *check_vars* table) for its Probe instance.

More advanced *Probes* can override base-class methods of *Probe* in particular `GeoHealthCheck.probe.Probe.perform_request()`. In that case the Probe-author should add one or more *GeoHealthCheck.result.Result* objects to *self.result* via *self.result.add_result(result)*

Writing a *Check* class requires providing the Plugin class-attributes (see above) including optional *PARAM_DEFS*. The actual check is implemented by overriding the *Check* base class method `GeoHealthCheck.check.Check.perform()`, setting the check-result via `GeoHealthCheck.check.Check.set_result()`.

Finally your Probes and Checks need to be made available to your GHC instance via *config_site.py* and need to be found on the Python-PATH of your app.

The above may seem daunting at first. Examples below will hopefully make things clear as writing new *Probes* and *Checks* may sometimes be a matter of minutes!

TODO: may need VERSION variable class-attr to support upgrades

3.5.3 Examples

GHC includes Probes and Checks that on first setup are made available in *config_site.py*. By studying the the GHC standard Probes and Checks under the subdir *GeoHealthCheck/plugins*, Plugin-authors may get a feel how implementation can be effected.

There are broadly two ways to write a *Probe*:

- using a *REQUEST_** class-attributes, i.e. letting GHC do the Probe's HTTP requests and checks
- overriding `GeoHealthCheck.probe.Probe.perform_request()`: making your own requests

An example for each is provided, including the *Checks* used.

The simplest Probe is one that does:

- an HTTP GET on a *Resource* URL
- checks if the HTTP Response is not errored, i.e. a 404 or 500 status
- optionally checks if the HTTP Response (not) contains expected strings

Below is the implementation of the class `GeoHealthCheck.plugins.probe.http.HttpGet`:

```
from GeoHealthCheck.probe import Probe


class HttpGet(Probe):
    """
    Do HTTP GET Request, to poll/ping any Resource bare url.
    """

    NAME = 'HTTP GET Resource URL'
    DESCRIPTION = 'Simple HTTP GET on Resource URL'
    RESOURCE_TYPE = '*:*'
    REQUEST_METHOD = 'GET'

```

(continues on next page)

(continued from previous page)

```
    CHECKS_AVAIL = {
        'GeoHealthCheck.plugins.check.checks.HttpStatusNoError': {
            'default': True
        },
        'GeoHealthCheck.plugins.check.checks.ContainsStrings': {},
        'GeoHealthCheck.plugins.check.checks.NotContainsStrings': {},
        'GeoHealthCheck.plugins.check.checks.HttpHasContentType': {}
    }
    """Checks avail"""

```

Yes, this is the entire implementation of `GeoHealthCheck.plugins.probe.http.HttpGet`! Only class-attributes are needed:

- standard Plugin attributes: *AUTHOR* ('GHC Team' by default) *NAME*, *DESCRIPTION*
- *RESOURCE_TYPE* = *'*:*'* denotes that any Resource may use this Probe (UI lists this Probe under "Probes Available" for Resource)
- *REQUEST_METHOD* = *'GET'* : GHC should use the HTTP GET request method
- *CHECKS_AVAIL* : all Check classes that can be applied to this Probe (UI lists these under "Checks Available" for Probe)

By setting:

```
'GeoHealthCheck.plugins.check.checks.HttpStatusNoError': {
   'default': True
},
```

that Check is automatically assigned to this Probe when created. The other Checks may be added and configured via the UI.

Next look at the Checks, the class `GeoHealthCheck.plugins.check.checks.HttpStatusNoError`:

```
import sys
from owslib.etree import etree
from GeoHealthCheck.plugin import Plugin
from GeoHealthCheck.check import Check
try:
    from html import escape  # python 3.x
except ImportError:
    from cgi import escape  # python 2.x


""" Contains basic Check classes for a Probe object."""


class HttpStatusNoError(Check):
    """
    Checks if HTTP status code is not in the 400- or 500-range.
    """

    NAME = 'HTTP status should not be errored'
    DESCRIPTION = 'Response should not contain a HTTP 400 or 500 range Error'

    def __init__(self):
        Check.__init__(self)
```

(continues on next page)

(continued from previous page)

```

    def perform(self):
        """Default check: Resource should at least give no error"""
        status = self.probe.response.status_code
        overall_status = status / 100
        if overall_status in [4, 5]:
            self.set_result(False, 'HTTP Error status=%d' % status)


class HttpHasHeaderValue(Check):
    """
```

Also this class is quite simple: providing class-attributes *NAME*, *DESCRIPTION* and implementing the base-class method `GeoHealthCheck.check.Check.perform()`. Via *self.probe* a Check always has a reference to its parent Probe instance and the HTTP Response object via *self.probe.response*. The check itself is a test if the HTTP status code is in the 400 or 500-range. The *CheckResult* is implicitly created by setting: *self.set_result(False, 'HTTP Error status=%d' % status)* in case of errors. *self.set_result()* only needs to be called when a Check fails. By default the Result is succes (*True*).

According to this pattern more advanced Probes are implemented for *OWS GetCapabilities*, the most basic test for OWS-es like WMS and WFS. Below the implementation of the class `GeoHealthCheck.plugins.probe.owsgetcaps.OwsGetCaps` and its derived classes for specific OWS-es:

```
from GeoHealthCheck.plugin import Plugin
from GeoHealthCheck.probe import Probe


class OwsGetCaps(Probe):
    """
    Fetch OWS capabilities doc
    """

    AUTHOR = 'GHC Team'
    NAME = 'OWS GetCapabilities'
    DESCRIPTION = 'Perform GetCapabilities Operation and check validity'
    # Abstract Base Class for OGC OWS GetCaps Probes
    # Needs specification in subclasses
    # RESOURCE_TYPE = 'OGC:ABC'

    REQUEST_METHOD = 'GET'
    REQUEST_TEMPLATE = \
        '?SERVICE={service}&VERSION={version}&REQUEST=GetCapabilities'

    PARAM_DEFS = {
        'service': {
            'type': 'string',
            'description': 'The OWS service within resource endpoint',
            'default': None,
            'required': True
        },
        'version': {
            'type': 'string',
            'description': 'The OWS service version within resource endpoint',
            'default': None,
            'required': True,
            'range': None
```

(continues on next page)

(continued from previous page)

```
        }
    }
    """Param defs, to be specified in subclasses"""

    CHECKS_AVAIL = {
        'GeoHealthCheck.plugins.check.checks.XmlParse': {
            'default': True
        },
        'GeoHealthCheck.plugins.check.checks.NotContainsOwsException': {
            'default': True
        },
        'GeoHealthCheck.plugins.check.checks.ContainsStrings': {
            'set_params': {
                'strings': {
                    'name': 'Contains Title Element',
                    'value': ['Title>']
                }
            },
            'default': True
        },
    }
    """
    Checks avail for all specific Caps checks.
    Optionally override Check PARAM_DEFS using set_params
    e.g. with specific `value`.
    """


class WmsGetCaps(OwsGetCaps):
    """Fetch WMS capabilities doc"""

    NAME = 'WMS GetCapabilities'
    RESOURCE_TYPE = 'OGC:WMS'

    PARAM_DEFS = Plugin.merge(OwsGetCaps.PARAM_DEFS, {

        'service': {
            'value': 'WMS'
        },
        'version': {
            'default': '1.1.1',
            'range': ['1.1.1', '1.3.0']
        }
    })
    """Param defs"""


class WfsGetCaps(OwsGetCaps):
    """WFS GetCapabilities Probe"""

    NAME = 'WFS GetCapabilities'
    RESOURCE_TYPE = 'OGC:WFS'

    def __init__(self):
        OwsGetCaps.__init__(self)

    PARAM_DEFS = Plugin.merge(OwsGetCaps.PARAM_DEFS, {
```

(continues on next page)

(continued from previous page)

```
        'service': {
            'value': 'WFS'
        },
        'version': {
            'default': '1.1.0',
            'range': ['1.0.0', '1.1.0', '2.0.2']
        }
    })
    """Param defs"""


class WcsGetCaps(OwsGetCaps):
    """WCS GetCapabilities Probe"""

    NAME = 'WCS GetCapabilities'
    RESOURCE_TYPE = 'OGC:WCS'

    PARAM_DEFS = Plugin.merge(OwsGetCaps.PARAM_DEFS, {
```

More elaborate but still only class-attributes are used! Compared to `GeoHealthCheck.plugins.probe.http.HttpGet`, two additional class-attributes are used in `GeoHealthCheck.plugins.probe.owsgetcaps.OwsGetCaps`:

- *REQUEST_TEMPLATE ='?SERVICE={service}&VERSION={version}&REQUEST=GetCapabilities'*
- *PARAM_DEFS* for the *REQUEST_TEMPLATE*

GHC will recognize a *REQUEST_TEMPLATE* (for GET or POST) and use *PARAM_DEFS* to substitute configured or default values, here defined in subclasses. This string is then appended to the Resource URL.

Three *Checks* are available, all included by default. Also see the construct:

```
'GeoHealthCheck.plugins.check.checks.ContainsStrings': {
   'set_params': {
       'strings': {
           'name': 'Contains Title Element',
           'value': ['Title>']
       }
   },
   'default': True
},
```

This not only assigns this Check automatically on creation, but also provides it with parameters, in this case a *Capabilities* response document should always contain a <*Title*> XML element. The class `GeoHealthCheck.plugins.check.checks.ContainsStrings` checks if a response doc contains all of a list (array) of configured strings. So the full checklist on the response doc is:

- is it XML-parsable: `GeoHealthCheck.plugins.check.checks.XmlParse`
- does not contain an Exception: `GeoHealthCheck.plugins.check.checks.NotContainsOwsException`
- does it have a <*Title*> element: `GeoHealthCheck.plugins.check.checks.ContainsStrings`

These Checks are performed in that order. If any fails, the Probe Run is in error.

We can now look at classes derived from `GeoHealthCheck.plugins.probe.owsgetcaps.OwsGetCaps`, in particular `GeoHealthCheck.plugins.probe.owsgetcaps.WmsGetCaps` and `GeoHealthCheck.plugins.probe.owsgetcaps.WfsGetCaps`. These only need to set their *RESOURCE_TYPE* e.g. *OGC:WMS* and override/merge *PARAM_DEFS*. For example for WMS:

```
PARAM_DEFS = Plugin.merge(OwsGetCaps.PARAM_DEFS, {

  'service': {
      'value': 'WMS'
  },
  'version': {
      'default': '1.1.1',
      'range': ['1.1.1', '1.3.0']
  }
})
```

This sets a fixed *value* for *service*, later becoming *service=WMS* in the URL request string. For *version* it sets both a *range* of values a user can choose from, plus a default value *1.1.1*. *Plugin.merge* needs to be used to merge-in new values. Alternatively *PARAM_DEFS* can be completely redefined, but in this case we only need to make per-OWS specific settings.

Also new in this example is parameterization of Checks for the class `GeoHealthCheck.plugins.check.checks.ContainsStrings`. This is a generic HTTP response checker for a list of strings that each need to be present in the response. Alternatively `GeoHealthCheck.plugins.check.checks.NotContainsStrings` has the reverse test. Both are extremely useful and for example available to our first example `GeoHealthCheck.plugins.probe.http.HttpGet`. The concept of *PARAM_DEFS* is the same for Probes and Checks.

In fact a Probe for any REST API could be defined in the above matter. For example, later in the project a Probe was added for the SensorThings API (STA), a recent OGC-standard for managing Sensor data via a JSON REST API. See the listing below:

```
from GeoHealthCheck.probe import Probe


class StaCaps(Probe):
    """Probe for SensorThings API main endpoint url"""

    NAME = 'STA Capabilities'
    DESCRIPTION = 'Perform STA Capabilities Operation and check validity'
    RESOURCE_TYPE = 'OGC:STA'

    REQUEST_METHOD = 'GET'

    def __init__(self):
        Probe.__init__(self)

    CHECKS_AVAIL = {
        'GeoHealthCheck.plugins.check.checks.HttpStatusNoError': {
            'default': True
        },
        'GeoHealthCheck.plugins.check.checks.JsonParse': {
            'default': True
        },
        'GeoHealthCheck.plugins.check.checks.ContainsStrings': {
            'default': True,
            'set_params': {
                'strings': {
                    'name': 'Must contain STA Entity names',
                    'value': ['Things', 'Datastreams', 'Observations',
                              'FeaturesOfInterest', 'Locations']
                }
```

(continues on next page)

(continued from previous page)

```
            }
        },
    }
    """
    Checks avail for all specific Caps checks.
    Optionally override Check.PARAM_DEFS using set_params
    e.g. with specific `value` or even `name`.
    """


class StaGetEntities(Probe):
    """Fetch STA entities of type and check result"""

    NAME = 'STA GetEntities'
    DESCRIPTION = 'Fetch all STA Entities of given type'
    RESOURCE_TYPE = 'OGC:STA'

    REQUEST_METHOD = 'GET'

    # e.g. http://52.26.56.239:8080/OGCSensorThings/v1.0/Things
    REQUEST_TEMPLATE = '/{entities}'

    def __init__(self):
        Probe.__init__(self)

    PARAM_DEFS = {
        'entities': {
            'type': 'string',
            'description': 'The STA Entity collection type',
            'default': 'Things',
            'required': True,
            'range': ['Things', 'DataStreams', 'Observations',
                      'Locations', 'Sensors', 'FeaturesOfInterest',
                      'ObservedProperties', 'HistoricalLocations']
        }
    }
    """Param defs"""

    CHECKS_AVAIL = {
        'GeoHealthCheck.plugins.check.checks.HttpStatusNoError': {
            'default': True
        },
        'GeoHealthCheck.plugins.check.checks.JsonParse': {
            'default': True
        }
    }
    """Check for STA Get entity Collection"""
```

Up to now all Probes were defined using and overriding class-attributes. Next is a more elaborate example where the Probe overrides the Probe baseclass method `GeoHealthCheck.probe.Probe.perform_request()`. The example is more of a showcase: `GeoHealthCheck.plugins.probe.wmsdrilldown.WmsDrilldown` literally drills-down through WMS-entities: starting with the *GetCapabilities* doc it fetches the list of *Layers* and does a *GetMap* on random layers etc. It uses *OWSLib.WebMapService*.

We show the first 70 lines here.

```python
import random

from GeoHealthCheck.probe import Probe
from GeoHealthCheck.result import Result
from owslib.wms import WebMapService


class WmsDrilldown(Probe):
    """
    Probe for WMS endpoint "drilldown": starting
    with GetCapabilities doc: get Layers and do
    GetMap on them etc. Using OWSLib.WebMapService.

    TODO: needs finalization.
    """

    NAME = 'WMS Drilldown'
    DESCRIPTION = 'Traverses a WMS endpoint by drilling down from Capabilities'
    RESOURCE_TYPE = 'OGC:WMS'

    REQUEST_METHOD = 'GET'

    PARAM_DEFS = {
        'drilldown_level': {
            'type': 'string',
            'description': 'How heavy the drilldown should be.',
            'default': 'minor',
            'required': True,
            'range': ['minor', 'moderate', 'full']
        }
    }
    """Param defs"""

    def __init__(self):
        Probe.__init__(self)

    def perform_request(self):
        """
        Perform the drilldown.
        See https://github.com/geopython/OWSLib/blob/
        master/tests/doctests/wms_GeoServerCapabilities.txt
        """
        wms = None

        # 1. Test capabilities doc, parses
        result = Result(True, 'Test Capabilities')
        result.start()
        try:
            wms = WebMapService(self._resource.url)
            title = wms.identification.title
            self.log('response: title=%s' % title)
        except Exception as err:
            result.set(False, str(err))

        result.stop()
        self.result.add_result(result)

```

(continues on next page)

(continued from previous page)

```
        # 2. Test layers
        # TODO: use parameters to work on less/more drilling
        # "full" could be all layers.
        result = Result(True, 'Test Layers')
        result.start()
        try:
            # Pick a random layer
            layer_name = random.sample(wms.contents.keys(), 1)[0]
            layer = wms[layer_name]

            # TODO Only use EPSG:4326, later random CRS
            if 'EPSG:4326' in layer.crsOptions \
                    and layer.boundingBoxWGS84:
```

This shows that any kind of simple or elaborate healthchecks can be implemented using single or multiple HTTP requests. As long as Result objects are set via *self.result.add_result(result)*. It is optional to also define *Checks* in this case. In the example `GeoHealthCheck.plugins.probe.wmsdrilldown.WmsDrilldown` example no Checks are used.

One can imagine custom Probes for many use-cases:

- drill-downs for OWS-es
- checking both the service and its metadata (CSW links in Capabilities doc e.g.)
- gaps in timeseries data (SOS, STA)
- even checking resources like a remote GHC itself!

Writing custom Probes is only limited by your imagination!

3.5.4 Configuration

Plugins available to a GHC installation are configured via *config_main.py* and overridden in *config_site.py*. By default all built-in Plugins are available.

- **GHC_PLUGINS**: *list* of built-in/core Plugin classes and/or modules available on installation
- **GHC_PROBE_DEFAULTS**: Default *Probe* class to assign on "add" per Resource-type
- **GHC_USER_PLUGINS**: *list* of your Plugin classes and/or modules available on installation

To add your Plugins, you need to configure **GHC_USER_PLUGINS**. In most cases you don't need to bother with **GHC_PLUGINS** and **GHC_PROBE_DEFAULTS**.

See an example for both below from *config_main.py* for **GHC_PLUGINS** and **GHC_PROBE_DEFAULTS**:

```
GHC_PLUGINS = [
    # Probes
    'GeoHealthCheck.plugins.probe.owsgetcaps',
    'GeoHealthCheck.plugins.probe.wms',
    'GeoHealthCheck.plugins.probe.wfs.WfsGetFeatureBbox',
    'GeoHealthCheck.plugins.probe.tms',
    'GeoHealthCheck.plugins.probe.http',
    'GeoHealthCheck.plugins.probe.sta',
    'GeoHealthCheck.plugins.probe.wmsdrilldown',

    # Checks
    'GeoHealthCheck.plugins.check.checks',
```

(continues on next page)

(continued from previous page)

```
]

# Default Probe to assign on "add" per Resource-type
GHC_PROBE_DEFAULTS = {
    'OGC:WMS': {
        'probe_class': 'GeoHealthCheck.plugins.probe.owsgetcaps.WmsGetCaps'
    },
    'OGC:WMTS': {
        'probe_class': 'GeoHealthCheck.plugins.probe.owsgetcaps.WmtsGetCaps'
    },
    'OSGeo:TMS': {
        'probe_class': 'GeoHealthCheck.plugins.probe.tms.TmsCaps'
    },
    'OGC:WFS': {
        'probe_class': 'GeoHealthCheck.plugins.probe.owsgetcaps.WfsGetCaps'
    },
    'OGC:WCS': {
        'probe_class': 'GeoHealthCheck.plugins.probe.owsgetcaps.WcsGetCaps'
    },
    'OGC:WPS': {
        'probe_class': 'GeoHealthCheck.plugins.probe.owsgetcaps.WpsGetCaps'
    },
    'OGC:CSW': {
        'probe_class': 'GeoHealthCheck.plugins.probe.owsgetcaps.CswGetCaps'
    },
    'OGC:SOS': {
        'probe_class': 'GeoHealthCheck.plugins.probe.owsgetcaps.SosGetCaps'
    },
    'OGC:STA': {
        'probe_class': 'GeoHealthCheck.plugins.probe.sta.StaCaps'
    },
    'urn:geoss:waf': {
        'probe_class': 'GeoHealthCheck.plugins.probe.http.HttpGet'
    },
    'WWW:LINK': {
        'probe_class': 'GeoHealthCheck.plugins.probe.http.HttpGet'
    },
    'FTP': {
        'probe_class': None
    }
}
```

To add your User Plugins these steps are needed:

- place your Plugin in any directory
- specify your Plugin in *config_site.py* in **GHC_USER_PLUGINS** var
- your Plugin module needs to be available in the *PYTHONPATH* of the GHC app

Let's say your Plugin is in file */plugins/ext/myplugin.py*. Example *config_site.py*

```
GHC_USER_PLUGINS='ext.myplugin'
```

Then you need to add the path */plugins* to the *PYTHONPATH* such that your Plugin is found.

3.5.5 User Plugins via Docker

The easiest way to add your Plugins (and running GHC in general!) is by using GHC Docker. See more info in the GHC Docker Plugins README.

3.5.6 Plugin API Docs

For GHC extension via Plugins the following classes apply.

Most Plugins have *PARAM_DEFS* parameter definitions. These are variables that should be filled in by the user in the GUI unless a fixed *value* applies.

Plugins - Base Classes

These are the base classes for GHC Plugins. Developers will mainly extend *Probe* and *Check*.

Results are helper-classes whose intances are generated by both *Probe* and *Check* classes. They form the ultimate outcome when running a *Probe*. A *ResourceResult* contains *ProbeResults*, the latter contains *CheckResults*.

class GeoHealthCheck.result.**CheckResult**(*check*, *check_vars*, *success=True*, *message='OK'*)
Bases: *GeoHealthCheck.result.Result*

Holds result data from a single Check.

class GeoHealthCheck.result.**ProbeResult**(*probe*, *probe_vars*)
Bases: *GeoHealthCheck.result.Result*

Holds result data from a single Probe: one Probe, N Checks.

class GeoHealthCheck.result.**ResourceResult**(*resource*)
Bases: *GeoHealthCheck.result.Result*

Holds result data from a single Resource: one Resource, N Probe(Results). Provides Run data.

class GeoHealthCheck.result.**Result**(*success=True*, *message='OK'*)
Bases: object

Base class for results for Resource or Probe.

Plugins - Probes

Probes apply to a single *Resource* instance. They are responsible for running requests against the Resource URL endpoint. Most *Probes* are implemented mainly via configuring class variables in particular *PARAM_DEFS* and *CHECKS_AVAIL*, but one is free to override any of the *Probe* baseclass methods.

Plugins - Checks

Checks apply to a single *Probe* instance. They are responsible for checking request results from their *Probe*.

3.6 License

3.7 Contact

The website geohealthcheck.org is the main entry point.

All development is done via GitHub: see https://github.com/geopython/geohealthcheck.

3.7.1 Links

- website: http://geohealthcheck.org
- GitHub: https://github.com/geopython/geohealthcheck
- Demo: https://demo.geohealthcheck.org
- Presentation: http://geohealthcheck.org/presentation
- Gitter Chat: https://gitter.im/geopython/GeoHealthCheck

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index