TeamEngine Integration Reflection and Notes

Forest Gafford ImageMattersLLC

The following is a summary about the work done on integrating OGC TeamEngine Validator. The health checker used to integrate with TeamEngine was forked from the GeoHealthCheck repo that can be found here:

<https://github.com/geopython/GeoHealthCheck>

TeamEngine health check integration repo:

<https://github.com/GeoPlatform/GeoHealthCheck/tree/feature/%2342-TEAM-Engine-monitoring-service>

GeoHealthCheck overview and setup:

Full documentation for GeoHealthCheck with explanations, examples, default settings, and configurations can be found at: <http://docs.geohealthcheck.org/en/latest/>

Generally speaking, a “resource” represents a service (like WFS or WMS), a “Probe” is a python class that is responsible for making a request to an external service, and a “Check” is a python class that interprets a given probe response and determines if the resource should pass or fail. A resource can have multiple probes and a probe can have multiple checks.

There is also an active Getter community for the GeoHealthCheck application that can help development: <https://gitter.im/geopython/GeoHealthCheck>

**Additional setup:**

An additional environment variable is required for running the application. The “STATIC\_HOME” environment variable has to be set. It is easiest to set this in the Docker file when building the container for deployment and running ‘export STATIC\_HOME=…” in bash on the development machine. The variable needs to point to the given directory:

[checkout root]/GeoHealthCheck/static/site

Example (local dev machine):

export STATIC\_HOME=/[path]/GeoHealthCheck/GeoHealthCheck/static/site/

Example (inside docker-compose.yml):

environment:

STATIC\_HOME: '/GeoHealthCheck/GeoHealthCheck/static/site/'

Structure of TeamEngine/GeoHealthCheck integration:

**sla.py**

The TeamEngine integration Probes can be found here:

<https://github.com/GeoPlatform/GeoHealthCheck/blob/feature/%2342-TEAM-Engine-monitoring-service/GeoHealthCheck/plugins/probe/sla.py>

[checkout root]/GeoHealthCheck/plugins/probe/sla.py

There are a few probes that are included in this file and they are all documented with inline code comments. The main thing to note here is that the Service Level Agreement (SLA) check, SLA\_compliance, implements its own request by overriding the ‘perform\_request’ function. This function is the focal point where the TeamEngine API is exercised and results are stored and passed onto the SLA checks.

At the time of demo completion the desired data to be displayed was HTML. This is returned from the TeamEngine API as a zip file. In this iteration the zip file was expanded under the static HTML directory to allow access to the HTML from the webserver. See the “Additional Setup” section above.

See:

<https://github.com/GeoPlatform/GeoHealthCheck/blob/feature/%2342-TEAM-Engine-monitoring-service/GeoHealthCheck/plugins/probe/sla.py#L153-L161>

and

<https://github.com/GeoPlatform/GeoHealthCheck/blob/feature/%2342-TEAM-Engine-monitoring-service/GeoHealthCheck/plugins/check/slachecks.py#L61-L63>

**slachecks.py**

TeamEngine integration Checks can be found here:

<https://github.com/GeoPlatform/GeoHealthCheck/blob/feature/%2342-TEAM-Engine-monitoring-service/GeoHealthCheck/plugins/check/slachecks.py>

[checkout root]/GeoHealthCheck/plugins/check/slachecks.py

The main SLA check is ‘SlaOGCTestValidation’ which is responsible for opening and determining test result outcomes. Because there in no machine-readable data in the HTML zip the system currently parses and pulls numbers from HTML for getting information about the test results. This is a fragile point in the demo!

**Notes:**

Due to changing requirements, a persistent field was added and never renamed. The “test\_xml” field on the results object is persisted to the database. As of the time this demo was completed the field held the name of the extracted zip directory of test results.

**Authenticated TeamEngine interactions branch:**

There was an early TeamEngine authentication feature request that was implemented and then abandoned. This branch expanded the TeamEngineAPI class to authenticate a user with TeamEngine and run tests as an authenticated user. The branch was in a working state at the time it was abandoned. The branch can be found here:

<https://github.com/GeoPlatform/GeoHealthCheck/tree/feature/run-TE-test-as-user>

Build and Deployment:

GeoHealthChecker can be built as a standalone Docker container and deployed anywhere. The two main options are to build a version of the container with custom Probes and Checks in it or to deploy a default image and use a volume mount for injecting custom Probes and Checks. This project was deployed using the former (built container) method. Both of these processes are documented in the GeoHealthCheck documentation:

<https://github.com/geopython/GeoHealthCheck/blob/master/docker/README.md>

Reflections on the project

Below is a list of some thoughts and observations when working with TeamEngine and GeoHealthChecker for this project.

GeoHealthChecker

GeoHealthChecker provides a flexible and easy framework for adding custom probes and checks for validating resources. The structure makes it so that you are able to extend the functionality of the application without making changes to the core. It also provides good documentation deploying a containerized build.

**Limitations of GeoHealthChecker:**

* Total probe time used for checking response:

The total time it takes to run all probes create the baseline response time. This means that the response time will include all time taken to run the entire TeamEngine test against a resource. This is obviously not a reliable way to determine actual response time for a resource.

* Hard to set some defaults and UI order:

The reflective processes engine is very nice and will generate the UI needed to customize probes and checks with user assigned variables. However, there are limitations to the available UI options that make the system a little less user friendly. The reflection far outweighs the UI shortcomings but may require more customization for future endeavors.

* Insufficient user/persistence/security model:

The system is not designed for security or scalability. A number of modifications were required in order to hide user resources from other users and allow for duplicate resources. In addition the default implementation is to use a local SQLite database that stores user credentials in plain text. There is an option to use PostgreSQL instead that would help some. Overall security is not a high priority for the system and that could be an issue in the future.

TeamEngine

The OGC TeamEngine validator seems to be in some very early stages and provides a number of limitations to overcome when integrating with it. The new API features are helpful for integrating but lack features that make it robust enough for a solid integration.

**Limitations of TeamEngine:**

* Does not work with HTTPS:

TeamEngine testes silently fail on all HTTPS resources. This is not documented anywhere that I was able to find. The HTTPS tests would silently fail without a useful error message. It was in a weekly meeting with OGC that it was understood the system does not support HTTPS endpoints. This does limit the number of resources that can be tested as some resources redirect to HTTPS by default.

* No error reporting:

TeamEngine has almost no error reporting. All errors returned from the engine were of this format: “Error executing test suite (wfs20-1.26)”. This made it very difficult to integrate with the system, especially from a programmatic standpoint.

* Inconsistent API parameters:

TeamEngine does not have a consistent way of running tests. While it is documented in plain HTML there is no way for a service to know the exact parameters that are required per test (no machine parse-able definition of the parameters). This requires an integrating service to perform awkward matching like this:  
<https://github.com/GeoPlatform/GeoHealthCheck/blob/feature/%2342-TEAM-Engine-monitoring-service/GeoHealthCheck/plugins/probe/sla.py#L118-L129>

* No easy parse-able TeamEngine Configuration:

There is plain text description from the API about what tests are loaded and the endpoints for running them. This complicated for a machine trying to discover the available tests. The API should support a machine-readable format as well (see bullet above this).

* Inconsistent data formats:

Various TeamEngine tests return different formats. Again, there is a way to discover the returned format but it is not mentioned in the API documentation, is not communicated by the API, and there is not a machine parse-able way of obtaining this information. The integration software is left on its own to determine the type of format (TestNG, CTL, etc.).

* Not history of test run:

The TeamEngine API does not provide a way of storing previously run tests from the API. This is a standard feature of other APIs that require user to use some sort of API key or other authentication method. This lack places the burden of storing test results on the integrating service when TeamEngine should already have a mechanism for doing this.

**Issues with the HTML zip results test results feature**

There are a few issues that are directly related to the new zipped HTML results format that should be mentioned:

* File structure in zipped files is inconsistent:

The file structure of the results returned from the API are different based on the test that is run. This requires the integrating software to search to find the root index.html file that’s deeply nested in files. This file should be at the top level of the extracted directory by default.

* Invalid zipped files returned on failure:

Invalid zip files are returned when there was an error in the system. The TeamEngine server responds with a 200 status code but returns an invalid file that cannot be unzipped. This requires additional error handling to be done in a linking service when the HTTP response code and returned data are there for reporting errors. It would be best to return a 5xx status code and no zip should be returned.

* No machine parse-able data in zipped HTML files:

The HTML files do not contain anything that a machine can read to get test results. The API only returns one format per test session. Some formats (xml) can parsed while others (HTML) cannot. This makes it difficult to report results as both human readable and machine readable are required. The current option is to either run the test twice to get both formats or attempt to parse HTML and use regex to capture values before converting them. A much easier approach would be to have an API key, a way to list different sessions, and a way to retrieve test results in a desired format without having to run repeated tests.