Gabbi Documentation

Release

Chris Dent

May 04, 2017

Contents

Gabbi tests are expressed in YAML as a series of HTTP requests with their expected response:

```
tests:
   - name: retrieve root
     GET: /
     status: 200
```

This will trigger a `GET` request to `/` on the configured *Target Host*. The test will pass if the response's status code is `200`.

CHAPTER 1

Test Structure

The top-level `tests` category contains an ordered sequence of test declarations, each describing the expected response to a given request:

Metadata

Key	Description	Notes
`name`	The test's name. Must be unique within a file.	**required**
`desc`	An arbitrary string describing the test.	
`verbose`	If `True` or `all` (synonymous), prints a representation of the current request and response to `stdout`, including both headers and body. If set to `headers` or `body`, only the corresponding part of the request and response will be printed. If the output is a TTY, colors will be used. If the body content-type is JSON it will be formatted for improved readability. See *`VerboseHttp`* for details.	defaults to `False`
`skip`	A string message which if set will cause the test to be skipped with the provided message.	defaults to `False`
`xfail`	Determines whether to expect this test to fail. Note that the test will be run anyway.	

Note: When tests are generated dynamically, the `TestCase` name will include the respective test's `name`, lowercased with spaces transformed to _. In at least some test runners this will allow you to select and filter on test name.

Request Parameters

Key	Description	Notes

any uppercase string	Any such key is considered an HTTP method, with the corresponding value expressing the URL.
	This is a shortcut combining `method` and `url` into a single statement:
	`GET: /index`
	corresponds to:
	`method: GET`
	`url: /index`	

`method`	The HTTP request method.	defaults to `GET`
`url`	The URL to request. This can either be a full path (e.g. "/index") or a fully qualified URL (i.e. including host and scheme, e.g. "http://example.org/index") — see *Target Host* for details.	Either this or the shortcut above is **required**
`request_headers`	A dictionary of key-value pairs representing request header names and values. These will be added to the constructed request.	
`query_parameters`	A dictionary of query parameters that will be added to the `url` as query string. If that URL already contains a set of query parameters, those wil be extended. See *Example Tests* for a demonstration of how the data is structured.	
`data`	A representation to pass as the body of a request. Note that `content-type` in `request_headers` should also be set — see *Data* for details.	
`redirects`	If `True`, redirects will automatically be followed.	defaults to `False`
`ssl`	Determines whether the request uses SSL (i.e. HTTPS). Note that the `url`'s scheme takes precedence if present — see *Target Host* for details.	defaults to `False`

Response Expectations

Key	Description	Notes
`status`	The expected response status code. Multiple acceptable response codes may be provided, separated by `||` (e.g. `302 || 301` — note, however, that this indicates ambiguity, which is generally undesirable).	defaults to `200`
`response_headers`	A dictionary of key-value pairs representing expected response header names and values. If a header's value is wrapped in `/.../`, it will be treated as a regular expression.	
`response_forbidden_headers`	A list of headers which must *not* be present.	
`response_strings`	A list of string fragments expected to be present in the response body.	
`response_json_paths`	A dictionary of JSONPath rules paired with expected matches. Using this rule requires that the content being sent from the server is JSON (i.e. a content type of `application/json` or containing `+json`) If the value is wrapped in `/.../` the result of the JSONPath query will be compared against the value as a regular expression.	
`poll`	A dictionary of two keys: • `count`: An integer stating the number of times to attempt this test before giving up. • `delay`: A floating point number of seconds to delay between attempts. This makes it possible to poll for a resource created via an asynchronous request. Use with caution.	

Note that many of these items allow *substitutions*.

Default values for a file's `tests` may be provided via the top-level `defaults` category. These take precedence over the global defaults (explained below).

For examples see the gabbi tests, *Example Tests* and the gabbi-demo tutorial.

CHAPTER 2

Fixtures

The top-level `fixtures` category contains a sequence of named *Fixtures*.

CHAPTER 3

Response Handlers

`response_*` keys are examples of Response Handlers. Custom handlers may be created by test authors for specific use cases. See *Content Handlers* for more information.

CHAPTER 4

Substitution

There are a number of magical variables that can be used to make reference to the state of a current test, the one just prior or any test prior to the current one. The variables are replaced with real values during test processing.

Global

- `$ENVIRON['<environment variable>']`: The name of an environment variable. Its value will replace the magical variable. If the string value of the environment variable is `"True"` or `"False"` then the resulting value will be the corresponding boolean, not a string.

Current Test

- `$SCHEME`: The current scheme/protocol (usually `http` or `https`).
- `$NETLOC`: The host and potentially port of the request.

Immediately Prior Test

- `$COOKIE`: All the cookies set by any `Set-Cookie` headers in the prior response, including only the cookie key and value pairs and no metadata (e.g. `expires` or `domain`).
- `$URL`: The URL defined in the prior request, after substitutions have been made. For backwards compatibility with earlier releases `$LAST_URL` may also be used, but if `$HISTORY` (see below) is being used, `$URL` must be used.
- `$LOCATION`: The location header returned in the prior response.
- `$HEADERS['<header>']`: The value of any header from the prior response.
- `$RESPONSE['<json path>']`: A JSONPath query into the prior response. See *JSONPath* for more on formatting.

Any Previous Test

- `$HISTORY['<test name>'].<magical variable expression>`: Any variable which refers to a prior test may be used in an expression that refers to any earlier test in the same file by identifying the target test by its name in a `$HISTORY` dictionary. For example, to refer to a value in a JSON object in the response of a test named `post json`:

  ```
  $HISTORY['post json'].$RESPONSE['$.key']
  ```

  This is a very powerful feature that could lead to test that are difficult for humans to read. Take care to optimize for the maintainers that will come after you, not yourself.

Note: Where a single-quote character, `'`, is shown in the variables above you may also use a double-quote character, `"`, but in any given expression the same character must be used at both ends.

All of these variables may be used in all of the following fields:

- `url`
- `query_parameters`
- `data`
- `request_headers`
- `response_strings`
- `response_json_paths` (on the value side of the key value pair)
- `response_headers` (on the value side of the key value pair)
- `response_forbidden_headers`
- `count` and `delay` fields of `poll`

With these variables it ought to be possible to traverse an API without any explicit statements about the URLs being used. If you need a replacement on a field that is not currently supported please raise an issue or provide a patch.

As all of these features needed to be tested in the development of gabbi itself, the gabbi tests are a good source of examples on how to use the functionality. See also *Example Tests* for a collection of examples and the gabbi-demo tutorial.

CHAPTER 5

Data

The `data` key has some special handing to allow for a bit more flexibility when doing a `POST` or `PUT`:

- If the value is not a string (that is, it is a sequence or structure) it is treated as a data structure that will be turned into a string by the `dumps` method on the relevant *content handler*. For example if the content-type of the body is `application/json` the data structure will be turned into a JSON string.
- If the value is a string that begins with `<@` then the rest of the string is treated as a filepath to be loaded. The path is relative to the test directory and may not traverse up into parent directories.
- If the value is an undecorated string, that's the value.

Note: When reading from a file care should be taken to ensure that a reasonable content-type is set for the data as this will control if any encoding is done of the resulting string value. If it is text, json, xml or javascript it will be encoded to UTF-8.

CHAPTER 6

Loading and Running Tests

To run gabbi tests with a test harness they must be generated in some fashion and then run. This is accomplished by a test loader. Initially gabbi only supported those test harnesses that supported the `load_tests` protocol in UnitTest. It now possible to also build and run tests with pytest with some limitations described below.

Note: It is also possible to run gabbi tests from the command line. See *YAML Runner*.

Warning: If test are being run with a runner that supports concurrency (such as `testrepository`) it is critical that the test runner is informed of how to group the tests into their respective suites. The usual way to do this is to use a regular expression that groups based on the name of the yaml files. For example, when using `testrepository` the `.testr.conf` file needs an entry similar to the following:

```
group_regex=gabbi\.suitemaker\.(test_[^_]+_[^_]+)
```

UnitTest Style Loader

To run the tests with a `load_tests` style loader a test file containing a `load_tests` method is required. That will look a bit like:

```
"""A sample test module."""

# For pathname munging
import os

# The module that build_tests comes from.
from gabbi import driver

# We need access to the WSGI application that hosts our service
from myapp import wsgiapp
```

```
# We're using fixtures in the YAML files, we need to know where to
# load them from.
from myapp.test import fixtures

# By convention the YAML files are put in a directory named
# "gabbits" that is in the same directory as the Python test file.
TESTS_DIR = 'gabbits'


def load_tests(loader, tests, pattern):
    """Provide a TestSuite to the discovery process."""
    test_dir = os.path.join(os.path.dirname(__file__), TESTS_DIR)
    # Pass "require_ssl=True" as an argument to force all tests
    # to use SSL in requests.
    return driver.build_tests(test_dir, loader,
                              intercept=wsgiapp.app,
                              fixture_module=fixtures)
```

For details on the arguments available when building tests see `build_tests()`.

Once the test loader has been created, it needs to be run. There are *many* options. Which is appropriate depends very much on your environment. Here are some examples using `unittest` or `testtools` that require minimal knowledge to get started.

By file:

```
python -m testtools.run -v test/test_loader.py
```

By module:

```
python -m testttols.run -v test.test_loader

python -m unittest -v test.test_loader
```

Using test discovery to locate all tests in a directory tree:

```
python -m testtools.run discover

python -m unittest discover test
```

See the source distribution and the tutorial repo for more advanced options, including using `testrepository` and `subunit`.

pytest

Since pytest does not support the `load_tests` system, a different way of generating tests is required. Two techniques are supported.

The original method (described below) used yield statements to generate tests which pytest would collect. This style of tests is deprecated as of `pytest>=3.0` so a new style using pytest fixtures has been developed.

pytest >= 3.0

In the newer technique, a test file is created that uses the `pytest_generate_tests` hook. Special care must be taken to always import the `test_pytest` method which is the base test that the pytest hook parametrizes to generate the tests from the YAML files. Without the method, the hook will not be called and no tests generated.

Here is a simple example file:

```
"""A sample pytest module for pytest >= 3.0."""

# For pathname munging
import os

# The module that py_test_generator comes from.
from gabbi import driver

# We need test_pytest so that pytest test collection works properly.
# Without this, the pytest_generate_tests method below will not be
# called.
from gabbi.driver import test_pytest  # noqa

# We need access to the WSGI application that hosts our service
from myapp import wsgiapp

# We're using fixtures in the YAML files, we need to know where to
# load them from.
from myapp.test import fixtures

# By convention the YAML files are put in a directory named
# "gabbits" that is in the same directory as the Python test file.
TESTS_DIR = 'gabbits'


def pytest_generate_tests(metafunc):
    test_dir = os.path.join(os.path.dirname(__file__), TESTS_DIR)
    driver.py_test_generator(
        test_dir, intercept=wsgiapp.app,
        fixture_module=fixtures, metafunc=metafunc)
```

This can then be run with the usual pytest commands. For example:

```
py.test -svx pytest3.0-example.py
```

pytest < 3.0

When using the older technique, test file must be created that calls *`py_test_generator()`* and yields the generated tests. That will look a bit like this:

```
"""A sample pytest module."""

# For pathname munging
import os

# The module that build_tests comes from.
from gabbi import driver

# We need access to the WSGI application that hosts our service
```

```python
from myapp import wsgiapp

# We're using fixtures in the YAML files, we need to know where to
# load them from.
from myapp.test import fixtures

# By convention the YAML files are put in a directory named
# "gabbits" that is in the same directory as the Python test file.
TESTS_DIR = 'gabbits'


def test_gabbits():
    test_dir = os.path.join(os.path.dirname(__file__), TESTS_DIR)
    # Pass "require_ssl=True" as an argument to force all tests
    # to use SSL in requests.
    test_generator = driver.py_test_generator(
        test_dir, intercept=wsgiapp.app,
        fixture_module=fixtures)

    for test in test_generator:
        yield test
```

This can then be run with the usual pytest commands. For example:

```
py.test -svx pytest-example.py
```

The older technique will continue to work with all versions of `pytest<4.0` but `>=3.0` will produce warnings. If you want to use the older technique but not see the warnings add `--disable-pytest-warnings` parameter to the invocation of `py.test`.

CHAPTER 7

Example Tests

What follows is a commented example of some tests in a single file demonstrating many of the *Test Format* features. See *Loading and Running Tests* for the Python needed to integrate with a testing harness.

```
# Fixtures can be used to set any necessary configuration, such as a
# persistence layer, and establish sample data. They operate per
# file. They are context managers, each one wrapping the next in the
# sequence.

fixtures:
    - ConfigFixture
    - SampleDataFixture

# There is an included fixture named "SkipAllFixture" which can be
# used to declare that all the tests in the given file are to be
# skipped.

# Each test file can specify a set of defaults that will be used for
# every request. This is useful for always specifying a particular
# header or always requiring SSL. These values will be used on every
# test in the file unless overriden. Lists and dicts are merged one
# level deep, except for "data" which is copied verbatim whether it
# is a string, list or dict (it can be all three).

defaults:
    ssl: True
    request_headers:
        x-my-token: zoom

# The tests themselves are a list under a "tests" key. It's useful
# to use plenty of whitespace to help readability.

tests:

# Each request *must* have a name which is unique to the file. When it
# becomes a TestCase the name will be lowercased and spaces will
```

```
# become "_". Use that generated name when limiting test runs.

    - name: a test for root
      desc: Some explanatory text that could be used by other tooling

# The URL can either be relative to a host specified elsewhere or
# be a fully qualified "http" or "https" URL. *You* are responsible
# for url-encoding the URL.

      url: /
      method: GET

# If no status or method are provided they default to "200" and
# "GET".

# Instead of explicitly stating "url" and "method" you can join
# those two keys into one key representing the method. The method
# *must* be uppercase.

    - name: another test for root
      desc: Same test as above but with GET key
      GET: /

# A single test can override settings in defaults (set above).

    - name: root without ssl redirects
      ssl: False
      GET: /
      status: 302

# When evaluating response headers it is possible to use a regular
# expression to not have to test the whole value.

      response_headers:
          location: /https/

# By default redirects will not be followed. This can be changed.

    - name: follow root without ssl redirect
      ssl: False
      redirects: True
      GET: /
      status: 200 # This is the response code after the redirect.

# URLs can express query parameters in two ways: either in the url
# value directly, or as query_parameters. If both are used then
# query_parameters are appended. In this example the resulting URL
# will be equivalent to
# /foo?section=news&article=1&article=2&date=yesterday
# but not necessarily in that order.

    - name: create a url with parameters
      GET: /foo?section=news
      query_parameters:
          article:
              - 1
              - 2
          date: yesterday
```

```
# Request headers can be used to declare media-type choices and
# experiment with authorization handling (amongst other things).
# Response headers allow evaluating headers in the response. These
# two together form the core value of gabbi.

    - name: test accept
      GET: /resource
      request_headers:
          accept: application/json
      response_headers:
          content-type: /application/json/

# If a header must not be present in a response at all that can be
# expressed in a test as follows.

    - name: test forbidden headers
      GET: /resource
      response_forbidden_headers:
          - x-special-header

# All of the above requests have defaulted to a "GET" method. When
# using "POST", "PUT" or "PATCH", the "data" key provides the
# request body.

    - name: post some text
      POST: /text_repo
      request_headers:
          content-type: text/plain
      data: "I'm storing this"
      status: 201

# If the data is not a string, it will be transformed into JSON.
# You must supply an appropriate content-type request header.

    - name: post some json
      POST: /json_repo
      request_headers:
          content-type: application/json
      data:
          name: smith
          abode: castle
      status: 201

# If the data is a string prepended with "<@" the value will be
# treated as the name of a file in the same directory as the YAML
# file. Again, you must supply an appropriate content-type. If the
# content-type is one of several "text-like" types, the content will
# be assumed to be UTF-8 encoded.

    - name: post an image
      POST: /image_repo
      request_headers:
          content-type: image/png
      data: <@kittens.png

# A single request can be marked to be skipped.
```

```yaml
    - name: patch an image
      skip: patching images not yet implemented
      PATCH: /image_repo/12d96fb8-e78c-11e4-8c03-685b35afa334

# Or a single request can be marked that it is expected to fail.

    - name: check allow headers
      desc: the framework doesn't do allow yet
      xfail: True
      PUT: /post_only_url
      status: 405
      response_headers:
          allow: POST

# The body of a response can be evaluated with response handlers.
# The most simple checks for a set of strings anywhere in the
# response. Note that the strings are members of a list.

    - name: check for css file
      GET: /blog/posts/12
      response_strings:
          - normalize.css

# For JSON responses, JSONPath rules can be used.

    - name: post some json get back json
      POST: /json_repo
      request_headers:
          content-type: application/json
      data:
          name: smith
          abode: castle
      status: 201
      response_json_paths:
          $.name: smith
          $.abode: castle

# Requests run in sequence. One test can make reference to the test
# immediately prior using some special variables.
# "$LOCATION" contains the "location" header in the previous
# response.
# "$HEADERS" is a pseudo dictionary containing all the headers of
# the previous response.
# "$ENVIRON" is a pseudo dictionary providing access to the current
# environment.
# "$RESPONSE" provides access to the JSON in the prior response, via
# JSONPath. See http://jsonpath-rw.readthedocs.io/ for
# jsonpath-rw formatting.
# $SCHEME and $NETLOC provide access to the current protocol and
# location (host and port).

    - name: get the thing we just posted
      GET: $LOCATION
      request_headers:
          x-magic-exchange: $HEADERS['x-magic-exchange']
          x-token: $ENVIRON['OS_TOKEN']
      response_json_paths:
          $.name: $RESPONSE['$.name']
```

```
        $.abode: $RESPONSE['$.abode']
    response_headers:
        content-location: /$SCHEME://$NETLOC/

# For APIs where resource creation is asynchronous it can be
# necessary to poll for the resulting resource. First we create the
# resource in one test. The next test uses the "poll" key to loop
# with a delay for a set number of times.

  - name: create asynch
    POST: /async_creator
    request_headers:
        content-type: application/json
    data:
        name: jones
        abode: bungalow
    status: 202

  - name: poll for created resource
    GET: $LOCATION
    poll:
        count: 10 # try up to ten times
        delay: .5 # wait .5 seconds between each try
    response_json_paths:
        $.name: $RESPONSE['$.name']
        $.abode: $RESPONSE['$.abode']
```

CHAPTER 8

JSONPath

Gabbi supports JSONPath both for validating JSON response bodies and within *substitutions*.

JSONPath expressions are provided by jsonpath_rw, with jsonpath_rw_ext custom extensions to address common requirements:

1. Sorting via `sorted` and `[/property]`.
2. Filtering via `[?property = value]`.
3. Returning the respective length via `len`.

(These apply both to arrays and key-value pairs.)

Here is a JSONPath example demonstrating some of these features. Given JSON data as follows:

```
{
    "pets": [
        {"type": "cat", "sound": "meow"},
        {"type": "dog", "sound": "woof"}
    ]
}
```

If the ordering of the list in `pets` is predictable and reliable it is relatively straightforward to test values:

```
response_json_paths:
    # length of list is two
    $.pets.`len`: 2
    # sound of second item in list is woof
    $.pets[1].sound: woof
```

If the ordering is *not* predictable additional effort is required:

```
response_json_paths:
    # sort by type
    $.pets[/type][0].sound: meow
    # sort by type, reversed
    $.pets[\type][0].sound: woof
```

```
    # all the sounds
    $.pets[/type]..sound: ['meow', 'woof']
    # filter by type = dog
    $.pets[?type = "dog"].sound: woof
```

If it is necessary to validate the entire JSON response use a JSONPath of `$`:

```
response_json_paths:
    $:
        pets:
            - type: cat
              sound: meow
            - type: dog
              sound: woof
```

This is not a technique that should be used frequently as it can lead to difficult to read tests and it also indicates that your gabbi tests are being used to test your serializers and data models, not just your API interactions.

There are more JSONPath examples in *Example Tests* and in the jsonpath_rw and jsonpath_rw_ext documentation.

CHAPTER 9

Target Host

The target host is the host on which the API to be tested can be found. Gabbi intends to preserve the flow and semantics of HTTP interactions as much as possible, and every HTTP request needs to be directed at a host of some form. Gabbi provides three ways to control this:

- Using wsgi-intercept to provide a fake socket and `WSGI` environment on an arbitrary host and port attached to a `WSGI` application (see intercept examples).
- Using fully qualified `url` values in the YAML defined tests (see full examples).
- Using a host and (optionally) port defined at test build time (see live examples).

The intercept and live methods are mutually exclusive per test builder, but either kind of test can freely intermix fully qualified URLs into the sequence of tests in a YAML file.

For test driven development and local tests the intercept style of testing lowers test requirements (no web server required) and is fast. Interception is performed as part of *Fixtures* processing as the most deeply nested fixture. This allows any configuration or database setup to be performed prior to the WSGI application being created.

For the implementation of the above see `build_tests()`.

CHAPTER 10

Fixtures

Each suite of tests is represented by a single YAML file, and may optionally use one or more fixtures to provide the necessary environment required by the tests in that file.

Fixtures are implemented as nested context managers. Subclasses of *GabbiFixture* must implement `start_fixture` and `stop_fixture` methods for creating and destroying, respectively, any resources managed by the fixture. While the subclass may choose to implement `__init__` it is important that no exceptions are thrown in that method, otherwise the stack of context managers will fail in unexpected ways. Instead initialization of real resources should happen in `start_fixture`.

At this time there is no mechanism for the individual tests to have any direct awareness of the fixtures. The fixtures exist, conceptually, on the server side of the API being tested.

Fixtures may do whatever is required by the testing environment, however there are two common scenarios:

- Establishing (and then resetting when a test suite has finished) any baseline configuration settings and persistence systems required for the tests.
- Creating sample data for use by the tests.

If a fixture raises `unittest.case.SkipTest` during `start_fixture` all the tests in the current file will be skipped. This makes it possible to skip the tests if some optional configuration (such as a particular type of database) is not available.

If an exception is raised while a fixture is being used, information about the exception will be stored on the fixture so that the `stop_fixture` method can decide if the exception should change how the fixture should clean up. The exception information can be found on `exc_type`, `exc_value` and `traceback` method attributes.

If an exception is raised when a fixture is started (in `start_fixture`) the first test in the suite using the fixture will be marked with an error using the traceback from the exception and all the tests in the suite will be skipped. This ensures that fixture failure is adequately captured and reported by test runners.

CHAPTER 11

Inner Fixtures

In some contexts (for example CI environments with a large number of tests being run in a broadly concurrent environment where output is logged to a single file) it can be important to capture and consolidate stray output that is produced during the tests and display it associated with an individual test. This can help debugging and avoids unusable output that is the result of multiple streams being interleaved.

Inner fixtures have been added to support this. These are fixtures more in line with the tradtional `unittest` concept of fixtures: a class on which `setUp` and `cleanUp` is automatically called.

`build_tests()` accepts a named parameter arguments of `inner_fixtures`. The value of that argument may be an ordered list of fixtures.Fixture classes that will be called when each individual test is set up.

An example fixture that could be useful is the FakeLogger.

Note: At this time `inner_fixtures` are not supported when using the pytest *loader*.

CHAPTER 12

Content Handlers

Content handlers are responsible for preparing request data and evaluating response data based on the content-type of the request and response. A content handler operates as follows:

- Structured YAML data provided via the `data` attribute is converted to a string or bytes sequence and used as request body.
- The response body (a string or sequence of bytes) is transformed into a content-type dependent structure and stored in an internal attribute named `response_data` that is:
  - used when evaluating the response body
  - used in `$RESPONSE[]` *substitutions*

By default, gabbi provides content handlers for JSON. In that content handler the `data` test key is converted from structured YAML into a JSON string. Response bodies are converted from a JSON string into a data structure in `response_data` that is used when evaluating `response_json_paths` entries in a test or doing JSONPath-based `$RESPONSE[]` substitutions.

Further content handlers can be added as extensions. Test authors may need these extensions for their own suites, or enterprising developers may wish to create and distribute extensions for others to use.

Note: One extension that is likely to be useful is a content handler that turns `data` into url-encoded form data suitable for POST and turns an HTML response into a DOM object.

Extensions

Content handlers are an evolution of the response handler concept in earlier versions gabbi. To preserve backwards compatibility with existing response handlers, old style response handlers are still allowed, but new handlers should implement the content handler interface (described below).

Registering additional custom handlers is done by passing a subclass of `ContentHandler` to `build_tests()`:

```
driver.build_tests(test_dir, loader, host=None,
                   intercept=simple_wsgi.SimpleWsgi,
                   content_handlers=[MyContentHandler])
```

If pytest is being used:

```
driver.py_test_generator(test_dir, intercept=simple_wsgi.SimpleWsgi,
                         content_handlers=[MyContenHandler])
```

Warning: When there are multiple handlers listed that accept the same content-type, the one that is earliest in the list will be used.

With `gabbi-run`, custom handlers can be loaded via the `--response-handler` option – see *load_response_handlers()* for details.

Note: The use of the `--response-handler` argument is done to preserve backwards compatibility and avoid excessive arguments. Both types of handler may be passed to the argument.

Implementation Details

Creating a content handler requires subclassing *ContentHandler* and implementing several methods. These methods are described below, but inspecting *JSONHandler* will be instructive in highlighting required arguments and techniques.

To provide a `response_<something>` response-body evaluator a subclass must define:

- `test_key_suffix`: This, along with the prefix `response_`, forms the key used in the test structure. It is a class level string.
- `test_key_value`: The key's default value, either an empty list (`[]`) or empty dict (`{}`). It is a class level value.
- `action`: An instance method which tests the expected values against the HTTP response - it is invoked for each entry, with the parameters depending on the default value. The arguments to `action` are (in order):
  - `self`: The current instance.
  - `test`: The currently active `HTTPTestCase`
  - `item`: The current entry if `test_key_value` is a list, otherwise the key half of the key/value pair at this entry.
  - `value`: `None` if `test_key_value` is a list, otherwise the value half of the key/value pair at this entry.

To translate request or response bodies to or from structured data a subclass must define an `accepts` method. This should return `True` if this class is willing to translate the provided content-type. During request processing it is given the value of the content-type header that will be sent in the request. During response processing it is given the value of the content-type header of the response. This makes it possible to handle different request and response bodies in the same handler, if desired. For example a handler might accept `application/x-www-form-urlencoded` and `text/html`.

If `accepts` is defined two additional static methods should be defined:

- `dumps`: Turn structured Python data from the `data` key in a test into a string or byte stream. The optional `test` param allows you to access the current test case which may help with manipulations for custom content handlers, e.g. `multipart/form-data` needs to add a `boundary` to the `Content-Type` header in order to mark the appropriate sections of the body.

- `loads`: Turn a string or byte stream in a response into a Python data structure. Gabbi will put this data on the `response_data` attribute on the test, where it can be used in the evaluations described above (in the `action` method) or in `$RESPONSE` handling. An example usage here would be to turn HTML into a DOM.

Finally if a `replacer` class method is defined, then when a `$RESPONSE` substitution is encountered, `replacer` will be passed the `response_data` of the prior test and the argument within the `$RESPONSE`.

Please see the JSONHandler source for additional detail.

CHAPTER 13

YAML Runner

If there is a running web service that needs to be tested and creating a test loader with *build_tests()* is either inconvenient or overkill it is possible to run YAML test files directly from the command line with the console-script `gabbi-run`. It accepts YAML on `stdin` or as multiple file arguments, and generates and runs tests and outputs a summary of the results.

The provided YAML may not use custom *Fixtures* but otherwise uses the default *Test Format*. *Target Host* information is either expressed directly in the YAML file or provided on the command line:

```
gabbi-run [host[:port]] < /my/test.yaml
```

or:

```
gabbi-run http://host:port < /my/test.yaml
```

To test with one or more files the following command syntax may be used:

```
gabbi-run http://host:port -- /my/test.yaml /my/other.yaml
```

Note: The filename arguments must come after a `--` and all other arguments (host, port, prefix, failfast) must come before the `--`.

Note: If files are provided, test output will use names including the name of the file. If any single file includes an error, the name of the file will be included in a summary of failed files at the end of the test report.

To facilitate using the same tests against the same application mounted in different locations in a WSGI server, a `prefix` may be provided as a second argument:

```
gabbi-run host[:port] [prefix] < /my/test.yaml
```

or in the target URL:

```
gabbi-run http://host:port/prefix < /my/test.yaml
```

The value of prefix will be prepended to the path portion of URLs that are not fully qualified.

Anywhere host is used, if it is a raw IPV6 address it should be wrapped in `[` and `]`.

If `https` is used in the target, then the tests in the provided YAML will default to `ssl: True`.

If a `-x` or `--failfast` argument is provided then `gabbi-run` will exit after the first test failure.

Use `-v` or `--verbose` with a value of `all`, `headers` or `body` to turn on *verbosity* for all tests being run.

CHAPTER 14

Release Notes

These are informal release notes for gabbi since version 1.0.0, highlighting major features and changes. For more detail see the commit logs on GitHub.

1.34.0

Substitutions in `$RESPONSE` handling now preserve numeric types instead of casting to a string. This is useful when servers are expecting strong types and tests want to send response data back to the server.

1.33.0

`count` and `delay` test keys allow *substitutions*. `gabbi.driver.build_tests()` accepts a `verbose` parameter to set test *verbosity* for an entire session.

1.32.0

Better failure reporting when using *gabbi-run* with multiple files. Test names are based on the files and a summary of failed files is provided at the end of the report.

1.31.0

Effectively capture a failure in a *fixture* and report the traceback. Without this some test runners swallow the error and discovering problems when developing fixtures can be quite challenging.

1.30.0

Thanks to Samuel Fekete, tests can use the `$HISTORY` dictionary to refer to any prior test in the same file, not just the one immediately prior, when doing *substitutions*.

1.29.0

Filenames used to read data into tests using the `<@` syntax may now use pathnames relative to the YAML file. See *Data*.

gabbi-run gains a –verbose parameter to force all tests run in a session to run with *verbose* set.

When using *pytest* to load tests, a new mechanism is available which avoids warnings produced in when using a version of pytest greater than `3.0`.

1.28.0

When verbosely displaying request and response bodies that are JSON, pretty print for improved readability.

1.27.0

Allow *gabbi-run* to accept multiple filenames as command line arguments instead of reading tests from stdin.

1.26.0

Switch from response handlers to *Content Handlers* to allow more flexible processing of both response _and_ request bodies.

Add *inner fixtures* for per test fixtures, useful for output capturing.

1.25.0

Allow the `test_loader_name` arg to *`gabbi.driver.build_tests()`* to override the prefix of the pretty printed name of generated tests.

1.24.0

String values in JSONPath matches may be wrapped in `/.../`` to be treated as regular expressions.

1.23.0

Better *documentation* of how to run gabbi in a concurrent environment. Improved handling of pytest fixtures and test counts.

1.22.0

Add `url` to *gabbi.driver.build_tests()* to use instead of `host`, `port` and `prefix`.

1.21.0

Add `require_ssl` to *gabbi.driver.build_tests()* to force use of SSL.

1.20.0

Add `$COOKIE` *substitution*.

1.19.1

Correctly support IPV6 hosts.

1.19.0

Add `$LAST_URL` *substitution*.

1.17.0

Introduce support for loading and running tests with pytest.

1.16.0

Use urllib3 instead of httplib2 for driving HTTP requests.

1.13.0

Add sorting and filtering to *JSONPath* handling.

1.11.0

Add the `response_forbidden_headers` to *response expectations*.

1.7.0

Instead of:

```
tests:
- name: a simple get
  url: /some/path
  method: get
```

1.7.0 also makes it possible to:

```
tests:
- name: a simple get
  GET: /some/path
```

Any upper case key is treated as a method.

1.4.0 and 1.5.0

Enhanced flexibility and colorization when setting tests to be *verbose*.

1.3.0

Adds the `query_parameters` key to *request parameters*.

1.2.0

The start of improvements and extensions to *JSONPath* handling. In this case the addition of the `len` function.

1.1.0

Vastly improved output and behavior in *gabbi-run*.

1.0.0

Version 1 was the first release with a commitment to a stable *Test Format*. Since then new fields have been added but have not been taken away.

CHAPTER 15

Contributors

The following people have contributed code to gabbi. Thanks to them. Thanks also to all the people who have made gabbi better by reporting issues and their successes and failures with using gabbi.

- Chris Dent
- FND
- Mehdi Abaakouk
- Tom Viner
- Jason Myers
- Ryan Spencer
- Kim Raymoure
- Travis Truman
- Samuel Fekete
- Michael McCune
- Imran Hayder
- Julien Danjou
- Danek Duvall
- Marc Abramowitz

CHAPTER 16

Frequently Asked Questions

Note: This section provides a collection of questions with answers that don't otherwise fit in the rest of the documentation. If something is missing, please create an issue.

As this document grows it will gain a more refined structure.

General

Is gabbi only for testing Python-based APIs?

No, you can use *gabbi-run* to test an HTTP service built in any programming language.

How do I run just one test?

Each YAML file contains a sequence of tests, each test within each file has a name. That name is translated to the name of the test by replacing spaces with an _.

When running tests that are *generated dynamically*, filtering based on the test name prior to the test being collected will not work in some test runners. Test runners that use a `--load-list` functionality can be convinced to filter after discovery.

pytest does this directly with the `-k` keyword flag.

When using testrepository with tox as used in gabbi's own tests it is possible to pass a filter in the tox command:

```
tox -epy27 -- get_the_widget
```

When using `testtools.run` and similar test runners it's a bit more complicated. It is necessary to provide the full name of the test as a list to `--load-list`:

```
python -m testtools.run --load-list \
    <(echo package.tests.test_api.yamlfile_get_the_widge.test_request)
```

Testing Style

Can I have variables in my YAML file?

Gabbi provides the `$ENVIRON` *substitution* which can operate a bit like variables that are set elsewhere and then used in the tests defined by the YAML.

If you find it necessary to have variables within a single YAML file you take advantage of YAML alias nodes list this:

```
vars:
  - &uuid_1 5613AABF-BAED-4BBA-887A-252B2D3543F8

tests:
- name: send a uuid to a post
  POST: /resource
  request_headers:
    content-type: application/json
  data:
    uuid: *uuid_1
```

You can alias all sorts of nodes, not just single items. Be aware that the replacement of an alias node happens while the YAML is being loaded, before gabbi does any processing.

How many tests should be put in one YAML file?

For the sake of readability it is best to keep each YAML file relatively short. Since each YAML file represents a sequence of requests, it usually makes sense to create a new file when a test is not dependent on any before it.

It's tempting to put all the tests for any resource or URL in the same file, but this eventually leads to files that are too long and are thus difficult to read.

CHAPTER 17

gabbi Package

`case` Module

A single HTTP request represented as a subclass of `testtools.TestCase`

The test case encapsulates the request headers and body and expected response headers and body. When the test is run an HTTP request is made using urllib3. Assertions are made against the reponse.

class `gabbi.case.`**`HTTPTestCase`**(**args*, ***kwargs*)

Bases: `testtools.testcase.TestCase`

Encapsulate a single HTTP request as a TestCase.

If the test is a member of a sequence of requests, ensure that prior tests are run.

To keep the test harness happy we need to make sure the setUp and tearDown are only run once.

`assert_in_or_print_output`(*expected*, *iterable*)

Assert the iterable contains expected or print some output.

If the output is long, it is limited by either GABBI_MAX_CHARS_OUTPUT in the environment or the MAX_CHARS_OUTPUT constant.

`base_test` **= {'status': '200', 'xfail': False, 'redirects': False, 'verbose': False, 'query_parameters': {}, 'url': '', 'skip': '**

`get_content_handler`(*content_type*)

Determine the content handler for this media type.

`replace_template`(*message*)

Replace magic strings in message.

`run`(*result=None*)

Store the current result handler on this test.

`setUp`()

`tearDown`()

test_request()
Run this request if it has not yet run.

If there is a prior test in the sequence, run it first.

gabbi.case.**potentialFailure**(*func*)
Decorate a test method that is expected to fail if 'xfail' is true.

driver Module

Generate HTTP tests from YAML files

Each HTTP request is its own TestCase and can be requested to be run in isolation from other tests. If it is a member of a sequence of requests, prior requests will be run.

A sequence is represented by an ordered list in a single YAML file.

Each sequence becomes a TestSuite.

An entire directory of YAML files is a TestSuite of TestSuites.

gabbi.driver.**build_tests**(*path*, *loader*, *host=None*, *port=8001*, *intercept=None*, *test_loader_name=None*, *fixture_module=None*, *response_handlers=None*, *content_handlers=None*, *prefix=''*, *require_ssl=False*, *url=None*, *inner_fixtures=None*, *verbose=False*)
Read YAML files from a directory to create tests.

Each YAML file represents an ordered sequence of HTTP requests.

Parameters

- **path** – The directory where yaml files are located.
- **loader** – The TestLoader.
- **host** – The host to test against. Do not use with `intercept`.
- **port** – The port to test against. Used with `host`.
- **intercept** – WSGI app factory for wsgi-intercept.
- **test_loader_name** – Base name for test classes. Use this to align the naming of the tests with other tests in a system.
- **fixture_module** – Python module containing fixture classes.
- **response_handers** – `ResponseHandler` classes.
- **content_handlers** (*List of ContentHandler classes.*) – ContentHandler classes.
- **prefix** – A URL prefix for all URLs that are not fully qualified.
- **url** – A full URL to test against. Replaces host, port and prefix.
- **require_ssl** – If `True`, make all tests default to using SSL.
- **inner_fixtures** (*List of fixtures.Fixture classes.*) – A list of `Fixtures` to use with each individual test request.
- **verbose** – If `True` or `'all'`, make tests verbose by default `'headers'` and `'body'` are also accepted.

Return type TestSuite containing multiple TestSuites (one for each YAML file) gabbi.driver.**py_test_generator**(*test_dir*, *host=None*, *port=8001*, *intercept=None*, *prefix=None*, *test_loader_name=None*, *fixture_module=None*, *response_handlers=None*, *content_handlers=None*, *require_ssl=False*, *url=None*, *metafunc=None*)

Generate tests cases for py.test

This uses build_tests to create TestCases and then yields them in a way that pytest can handle.

gabbi.driver.**test_pytest**(*test*, *result*)

gabbi.driver.**test_suite_from_yaml**(*loader*, *test_base_name*, *test_yaml*, *test_directory*, *host*, *port*, *fixture_module*, *intercept*, *prefix=''*)

Legacy wrapper retained for backwards compatibility.

`suitemaker` Module

The code that creates a suite of tests.

The key piece of code is `test_suite_from_dict()`. It produces a `gabbi.suite.GabbiSuite` containing one or more `gabbi.case.HTTPTestCase`.

class gabbi.suitemaker.**TestBuilder**

Bases: `type`

Metaclass to munge a dynamically created test.

required_attributes = {'has_run': False}

class gabbi.suitemaker.**TestMaker**(*test_base_name*, *test_defaults*, *test_directory*, *fixture_classes*, *loader*, *host*, *port*, *intercept*, *prefix*, *response_handlers*, *content_handlers*, *test_loader_name=None*, *inner_fixtures=None*)

Bases: `object`

A class for encapsulating test invariants.

All of the tests in a single gabbi file have invariants which are provided when creating each HTTPTestCase. It is not useful to pass these around when making each test case. So they are wrapped in this class which then has make_one_test called multiple times to generate all the tests in the suite.

make_one_test(*test_dict*, *prior_test*)

Create one single HTTPTestCase.

The returned HTTPTestCase is added to the TestSuite currently being built (one per YAML file).

gabbi.suitemaker.**test_suite_from_dict**(*loader*, *test_base_name*, *suite_dict*, *test_directory*, *host*, *port*, *fixture_module*, *intercept*, *prefix=''*, *handlers=None*, *test_loader_name=None*, *inner_fixtures=None*)

Generate a GabbiSuite from a dict represent a list of tests.

The dict takes the form:

Parameters

- **fixtures** – An optional list of fixture classes that this suite can use.
- **defaults** – An optional dictionary of default values to be used in each test.
- **tests** – A list of individual tests, themselves each being a dictionary. See `gabbi.case.BASE_TEST`.

gabbi.suitemaker.**test_update**(*orig_dict*, *new_dict*)
Modify test in place to update with new data.

fixture Module

Manage fixtures for gabbi at the test suite level.

class gabbi.fixture.**GabbiFixture**
Bases: object

A context manager that operates as a fixture.

Subclasses must implement start_fixture and stop_fixture, each of which contain the logic for stopping and starting whatever the fixture is. What a fixture is is left as an exercise for the implementor.

These context managers will be nested so any actual work needs to happen in start_fixture and stop_fixture and not in __init__. Otherwise exception handling will not work properly.

start_fixture()
Implement the actual workings of starting the fixture here.

stop_fixture()
Implement the actual workings of stopping the fixture here.

exception gabbi.fixture.**GabbiFixtureError**
Bases: exceptions.Exception

Generic exception for GabbiFixture.

class gabbi.fixture.**SkipAllFixture**
Bases: *gabbi.fixture.GabbiFixture*

A fixture that skips all the tests in the current suite.

start_fixture()

gabbi.fixture.**nest**(**args*, ***kwds*)
Nest a series of fixtures.

This is duplicated from nested in the stdlib, which has been deprecated because of issues with how exceptions are difficult to handle during __init__. Gabbi needs to nest an unknown number of fixtures dynamically, so the with syntax that replaces nested will not work.

handlers Module

Package for response and content handlers that process the body of a response in various ways.

handlers.base Module

Base classes for response and content handlers.

class gabbi.handlers.base.**ContentHandler**
Bases: *gabbi.handlers.base.ResponseHandler*

A subclass of ResponseHandlers that adds content handling.

static accepts(*content_type*)
Return True if this handler can handler this type.

static dumps (*data*, *pretty=False*, *test=None*)
Return structured data as a string.

If pretty is true, prettify.

static loads (*data*)
Create structured (Python) data from a stream.

classmethod replacer (*response_data*, *path*)
Return the string that is replacing RESPONSE.

class `gabbi.handlers.base.`**`ResponseHandler`**
Bases: `object`

Add functionality for making assertions about an HTTP response.

A subclass may implement two methods: `action` and `preprocess`.

`preprocess` takes one argument, the `TestCase`. It is called exactly once for each test before looping across the assertions. It is used, rarely, to copy the `test.output` into a useful form (such as a parsed DOM).

`action` takes two or three arguments. If `test_key_value` is a list `action` is called with the test case and a single list item. If `test_key_value` is a dict then `action` is called with the test case and a key and value pair.

`action` (*test*, *item*, *value=None*)
Test an individual entry for this response handler.

If the entry is a key value pair the key is in item and the value in value. Otherwise the entry is considered a single item from a list.

`preprocess` (*test*)
Do any pre-single-test preprocessing.

`test_key_suffix` = **‘’**

`test_key_value` = **[]**

`handlers.core` Module

Core response handlers.

class `gabbi.handlers.core.`**`ForbiddenHeadersResponseHandler`**
Bases: *`gabbi.handlers.base.ResponseHandler`*

Test that listed headers are not in the response.

`action` (*test*, *forbidden*, *value=None*)

`test_key_suffix` = **‘forbidden_headers’**

`test_key_value` = **[]**

class `gabbi.handlers.core.`**`HeadersResponseHandler`**
Bases: *`gabbi.handlers.base.ResponseHandler`*

Compare expected headers with actual headers.

If a header value is wrapped in `/` it is treated as a raw regular expression.

Headers values are always treated as strings.

`action` (*test*, *header*, *value=None*)

`test_key_suffix` = **‘headers’**

test_key_value = {}

class gabbi.handlers.core.**StringResponseHandler**

Bases: *gabbi.handlers.base.ResponseHandler*

Test for matching strings in the the response body.

action (*test*, *expected*, *value=None*)

test_key_suffix = 'strings'

test_key_value = []

handlers.jsonhandler Module

JSON-related content handling.

class gabbi.handlers.jsonhandler.**JSONHandler**

Bases: *gabbi.handlers.base.ContentHandler*

A ContentHandler for JSON

- Structured test `data` is turned into JSON when request content-type is JSON.
- Response bodies that are JSON strings are made into Python data on the test `response_data` attribute when the response content-type is JSON.
- A `response_json_paths` response handler is added.
- JSONPaths in $RESPONSE substitutions are supported.

static **accepts** (*content_type*)

action (*test*, *path*, *value=None*)

Test json_paths against json data.

static **dumps** (*data*, *pretty=False*, *test=None*)

static **extract_json_path_value** (*data*, *path*)

Extract the value at JSON Path path from the data.

The input data is a Python datastructure, not a JSON string.

static **loads** (*data*)

classmethod **replacer** (*response_data*, *match*)

test_key_suffix = 'json_paths'

test_key_value = {}

suite Module

A TestSuite for containing gabbi tests.

This suite has two features: the contained tests are ordered and there are suite-level fixtures that operate as context managers.

class gabbi.suite.**GabbiSuite** (*tests=()*)

Bases: `unittest.suite.TestSuite`

A TestSuite with fixtures.

The suite wraps the tests with a set of nested context managers that operate as fixtures.

If a fixture raises unittest.case.SkipTest during setup, all the tests in this suite will be skipped.

run (*result*, *debug=False*)
Override TestSuite run to start suite-level fixtures.

To avoid exception confusion, use a null Fixture when there are no fixtures.

start (*result*)
Start fixtures when using pytest.

stop ()
Stop fixtures when using pytest.

gabbi.suite.**noop** (**args*)
A noop method used to disable collected tests.

runner Module

Implementation of a command-line runner for gabbi files (AKA suites).

gabbi.runner.**extract_file_paths** (*argv*)
Extract command-line arguments following the – end-of-options delimiter, if any.

gabbi.runner.**initialize_handlers** (*response_handlers*)

gabbi.runner.**load_response_handlers** (*import_path*)
Load and return custom response handlers from the given Python package or module.

The import path references either a specific response handler class ("package.module:class") or a module that contains one or more response handler classes ("package.module").

For the latter, the module is expected to contain a `gabbi_response_handlers` object, which is either a list of response handler classes or a function returning such a list.

gabbi.runner.**run** ()
Run simple tests from STDIN.

This command provides a way to run a set of tests encoded in YAML that is provided on STDIN. No fixtures are supported, so this is primarily designed for use with real running services.

Host and port information may be provided in three different ways:

- In the URL value of the tests.
- In a *host* or *host:port* argument on the command line.
- In a URL on the command line.

An example run might looks like this:

```
gabbi-run example.com:9999 < mytest.yaml
```

or:

```
gabbi-run http://example.com:999 < mytest.yaml
```

It is also possible to provide a URL prefix which can be useful if the target application might be mounted in different locations. An example:

```
gabbi-run example.com:9999 /mountpoint < mytest.yaml
```

or:

```
gabbi-run http://example.com:9999/mountpoint < mytest.yaml
```

Use *-x* or *–failfast* to abort after the first error or failure:

```
gabbi-run -x example.com:9999 /mountpoint < mytest.yaml
```

Use *-v* or *–verbose* with a value of *all*, *headers* or *body* to turn on verbosity for all tests being run.

Multiple files may be named as arguments, separated from other arguments by a `--`. Each file will be run as a separate test suite:

```
gabbi-run http://example.com -- /path/to/x.yaml /path/to/y.yaml
```

Output is formatted as unittest summary information.

`gabbi.runner.`**`run_suite`**(*handle*, *handler_objects*, *host*, *port*, *prefix*, *force_ssl=False*, *failfast=False*, *data_dir='.'*, *verbosity=False*, *name='input'*)
Run the tests from the YAML in handle.

`reporter` Module

TestRunner and TestResult for gabbi-run.

class `gabbi.reporter.`**`ConciseTestResult`**(*stream*, *descriptions*, *verbosity*)
Bases: `unittest.runner.TextTestResult`

A TextTestResult with simple but useful output.

If the output is a tty or GABBI_FORCE_COLOR is set in the environment, output will be colorized.

`addError`(*test*, *err*)

`addExpectedFailure`(*test*, *err*)

`addFailure`(*test*, *err*)

`addSkip`(*test*, *reason*)

`addSuccess`(*test*)

`addUnexpectedSuccess`(*test*)

`getDescription`(*test*)

`printErrorList`(*flavor*, *errors*)

`startTest`(*test*)

class `gabbi.reporter.`**`ConciseTestRunner`**(*stream=<open file '<stderr>', mode 'w'>*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*)
Bases: `unittest.runner.TextTestRunner`

A TextTestRunner that uses ConciseTestResult for reporting results.

`resultclass`
alias of `ConciseTestResult`

class `gabbi.reporter.`**`PyTestResult`**(*stream=None*, *descriptions=None*, *verbosity=None*)
Bases: `unittest.result.TestResult`

Wrap a test result to allow it to work with pytest.

The main behaviors here are:

- to turn what had been exceptions back into exceptions
- use pytest's skip and xfail methods

`addError`(*test*, *err*)

`addExpectedFailure`(*test*, *err*)

`addFailure`(*test*, *err*)

`addSkip`(*test*, *reason*)

`utils` Module

Utility functions grab bag.

`gabbi.utils.`**`create_url`**(*base_url*, *host*, *port=None*, *prefix=''*, *ssl=False*)
Given pieces of a path-based url, return a fully qualified url.

`gabbi.utils.`**`decode_response_content`**(*header_dict*, *content*)
Decode content to a proper string.

`gabbi.utils.`**`extract_content_type`**(*header_dict*, *default='application/binary'*)
Extract parsed content-type from headers.

`gabbi.utils.`**`get_colorizer`**(*stream*)
Return a function to colorize a string.

Only if stream is a tty .

`gabbi.utils.`**`host_info_from_target`**(*target*, *prefix=None*)
Turn url or host:port and target into test destination.

`gabbi.utils.`**`load_yaml`**(*handle=None*, *yaml_file=None*)
Read and parse any YAML file or filehandle.

Let exceptions flow where they may.

If no file or handle is provided, read from STDIN.

`gabbi.utils.`**`not_binary`**(*content_type*)
Decide if something is content we'd like to treat as a string.

`gabbi.utils.`**`parse_content_type`**(*content_type*, *default_charset='utf-8'*)
Parse content type value for media type and charset.

`exception` Module

Gabbi specific exceptions.

exception `gabbi.exception.`**`GabbiFormatError`**
Bases: `exceptions.ValueError`

An exception to encapsulate poorly formed test data.

exception `gabbi.exception.`**`GabbiSyntaxWarning`**
Bases: `exceptions.SyntaxWarning`

A warning about syntax that is not desirable.

`httpclient` Module

Subclass of Http class for verbosity.

class `gabbi.httpclient.`**`Http`**(*num_pools=10*, *headers=None*, ***connection_pool_kw*)
Bases: `urllib3.poolmanager.PoolManager`

A subclass of the urllib3.PoolManager to munge the data.

This transforms the response to look more like what httplib2 provided when it was used as the httpclient.

`request`(*absolute_uri*, *method*, *body*, *headers*, *redirect*)

class `gabbi.httpclient.`**`VerboseHttp`**(***kwargs*)
Bases: `gabbi.httpclient.Http`

A subclass of Http that verbosely reports on activity.

If the output is a tty or `GABBI_FORCE_COLOR` is set in the environment, then output will be colorized according to `COLORMAP`.

Output can include request and response headers, request and response body content (if of a printable content-type), or both.

The color of the output has reasonable defaults. These may be overridden by setting the following environment variables

- GABBI_CAPTION_COLOR
- GABBI_HEADER_COLOR
- GABBI_REQUEST_COLOR
- GABBI_STATUS_COLOR

to any of: BLACK RED GREEN YELLOW BLUE MAGENTA CYAN WHITE

`COLORMAP` = {'status': 'CYAN', 'caption': 'BLUE', 'request': 'CYAN', 'header': 'YELLOW'}

`HEADER_BLACKLIST` = ['status', 'reason']

`REQUEST_PREFIX` = '>'

`RESPONSE_PREFIX` = '<'

`request`(*absolute_uri*, *method*, *body*, *headers*, *redirect*)
Display request parameters before requesting.

`gabbi.httpclient.`**`get_http`**(*verbose=False*, *caption=''*)
Return an Http class for making requests.

`json_parser` Module

Keep one single global jsonpath parser.

`gabbi.json_parser.`**`parse`**(*path*)
Parse a JSONPath expression use the global parser.

CHAPTER 18

Gabbi

Gabbi is a tool for running HTTP tests where requests and responses are expressed as declarations in a collection of YAML files. The simplest test looks like this:

```
tests:
- name: A test
  GET: /api/resources/id
```

See the rest of these docs for more details on the many features and formats for setting request headers and bodies and evaluating responses.

The name is derived from "gabby": excessively talkative. In a test environment having visibility of what a test is actually doing is a good thing. This is especially true when the goal of a test is to test the HTTP, not the testing infrastructure. Gabbi tries to put the HTTP interaction in the foreground of testing.

Tests can be run using *unittest* style test runners or py.test or from the command line with a *gabbi-run* script.

If you want to get straight to creating tests look at *Example Tests*, the test files in the source distribution and *Test Format*. A gabbi-demo repository provides a tutorial of using gabbi to build an API, via the commit history of the repo.

Purpose

Gabbi works to bridge the gap between human readable YAML files (see *Test Format* for details) that represent HTTP requests and expected responses and the rather complex world of automated testing.

Each YAML file represents an ordered list of HTTP requests along with the expected responses. This allows a single file to represent a process in the API being tested. For example:

- Create a resource.
- Retrieve a resource.
- Delete a resource.
- Retrieve a resource again to confirm it is gone.

At the same time it is still possible to ask gabbi to run just one request. If it is in a sequence of tests, those tests prior to it in the YAML file will be run (in order). In any single process any test will only be run once. Concurrency is handled such that one file runs in one process.

These features mean that it is possible to create tests that are useful for both humans (as tools for learning, improving and developing APIs) and automated CI systems.

Significant flexibility and power is available in the *Test Format* to make it relatively straightforward to test existing complex APIs. This extended functionality includes the use of JSONPath to query response bodies and templating of test data to allow access to the prior HTTP response in the current request. For APIs which do not use JSON additional *Content Handlers* can be created.

Care should be taken when using this functionality when you are creating a new API. If your API is so complex that it needs complex test files then you may wish to take that as a sign that your API itself too complex. One goal of gabbi is to encourage transparent and comprehensible APIs.

Though gabbi is written in Python and under the covers uses `unittest` data structures and processes, there is no requirement that the *Target Host* be a Python-based service. Anything talking HTTP can be tested. A *YAML Runner* makes it possible to simply create YAML files and point them at a running server.

Python Module Index

g

Index

H

I

J

L

M

N

P

R

S

T

V