# Contents

1  Why Tavern  
   
2  Contents  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Why Tavern</td>
</tr>
<tr>
<td>2</td>
<td>Contents</td>
</tr>
</tbody>
</table>
Tavern is an advanced pytest based API testing framework for HTTP, MQTT or other protocols.

Note that Tavern only supports Python 3.4 and up. At the time of writing we test against Python 3.4-3.7 and pypy3. Python 2 is now unsupported.
Choosing an API testing framework can be tough. Tavern was started in 2017 to address some of our concerns with other testing frameworks.

In short, we think the best things about Tavern are:

1.1 **It’s Lightweight.**

Tavern is a small codebase which uses pytest under the hood.

1.2 **Easy to Write, Easy to Read and Understand.**

The yaml syntax allows you to abstract what you need with anchors, whilst using `pytest.mark` to organise your tests. Your tests should become more maintainable as a result.

1.3 **Test Anything**

From the simplest API test through to the most complex of requests, tavern remains readable and easy to extend. We’re aiming for developers to not need the docs open all the time!

1.4 **Extensible**

Almost all common test usecases are covered, but for everything else it’s very easy to drop in to python/pytest to extend. Use fixtures, hooks and things you already know.
1.5 Growing Ecosystem

Tavern is still in active development and is used by 100s of companies.
2.1 Basic Concepts

2.1.1 Anatomy of a test

Tests are defined in YAML with a **test_name**, one or more **stages**, each of which has a **name**, a **request** and a **response**. Taking the simple example:

```yaml
test_name: Get some fake data from the JSON placeholder API
stages:
  - name: Make sure we have the right ID
    request:
      url: https://jsonplaceholder.typicode.com/posts/1
      method: GET
    response:
      status_code: 200
      json:
        id: 1
    save:
      json:
        returned_id: id
```

If using the pytest plugin (the recommended way of using Tavern), this needs to be in a file called `test_x.tavern.yaml`, where `x` should be a description of the contained tests.

If you want to call your files something different (though this is not recommended) it is also possible to specify a custom regular expression to match filenames. For example, if you want to call all of your files `tavern_test_x.yaml, tavern_test_y.yaml, etc.` then use the `tavern-file-path-regex` option in the configuration file or on the command line. For example, `py.test --tavern-file-path-regex "tavern_test_.*\.yaml"`

**test_name** is, as expected, the name of that test. If the pytest plugin is being used to run integration tests, this is what the test will show up as in the pytest report, for example:
This can then be selected with the \texttt{-k} flag to pytest \texttt{-e} \textit{pass pytest -k fake} to run all tests with ‘fake’ in the name.

\textbf{stages} is a list of the stages that make up the test. A simple test might just be to check that an endpoint returns a 401 with no login information. A more complicated one might be:

1. Log in to server
   - \texttt{POST} login information in body
   - Expect login details to be returned in body
1. Get user information
   - \texttt{GET} with login information in \texttt{Authorization} header
   - Expect user information returned in body
1. Create a new resource with that user information
   - \texttt{POST} with login information in \texttt{Authorization} header and user information in body
   - Expect a 201 with the created resource in the body
1. Make sure it’s stored on the server
   - \texttt{GET} with login information in \texttt{Authorization} header
   - Expect the same information returned as in the previous step

The \textbf{name} of each stage is a description of what is happening in that particular test.

\textbf{Request}

The \textbf{request} describes what will be sent to the server. The keys for this are passed directly to the \texttt{requests} library (after preprocessing) - at the moment the only supported keys are:

- \texttt{url} - a string, including the protocol, of the address of the server that will be queried
- \texttt{json} - a mapping of (possibly nested) key: value pairs/lists that will be converted to JSON and sent as the request body.
- \texttt{params} - a mapping of key: value pairs that will go into the query parameters.
- \texttt{data} - Either a mapping of key: value pairs that will go into the body as application/x-www-url-formencoded data, or a string that will be sent by itself (with no content-type).
- \texttt{headers} - a mapping of key: value pairs that will go into the headers. Defaults to adding a \texttt{content-type: application/json} header.
- \texttt{method} - one of \texttt{GET}, \texttt{POST}, \texttt{PUT}, \texttt{DELETE}, \texttt{PATCH}, \texttt{OPTIONS}, or \texttt{HEAD}. Defaults to \texttt{GET} if not defined

For more information, refer to the \texttt{requests} documentation.

\textbf{Response}

The \textbf{response} describes what we expect back. There are a few keys for verifying the response:

- \texttt{status_code} - an integer corresponding to the status code that we expect, or a list of status codes if you are expecting one of a few status codes. Defaults to 200 if not defined.
• **json** - Assuming the response is json, check the body against the values given. Expects a mapping (possibly nested) key: value pairs/lists. This can also use an external check function, described further down.

• **headers** - a mapping of key: value pairs that will be checked against the headers.

• **redirect_query_params** - Checks the query parameters of a redirect url passed in the location header (if one is returned). Expects a mapping of key: value pairs. This can be useful for testing implementation of an OpenID connect provider, where information about the request may be returned in redirect query parameters.

The **save** block can save values from the response for use in future requests. Things can be saved from the body, headers, or redirect query parameters. When used to save something from the json body, this can also access dictionaries and lists recursively. If the response is:

```json
"thing":
  "nested":
    1, 2, 3, 4
```

This can be saved into the value `first_val` with this response block:

```py
response:
  save:
    json:
      first_val: thing.nested[0]
```

The query should be defined as a JMES query (see JMESPath for more information). In the above example, this essentially performs the operation `json["thing"]["nested"][0]`. This can be used to perform powerful queries on response data, but note that only ‘simple’ values like integers, strings, or float values can be saved. Trying to save a ‘block’ of data such as a JSON list or object is currently unsupported and will cause the test to fail.

**NOTE:** The behaviour of these queries used to be different and indexing into an array was done like `thing.nested.0`. This will be deprecated in the 1.0 release.

It is also possible to save data using function calls, **explained below**.

For a more formal definition of the schema that the tests are validated against, check `tests schema` in the main Tavern repository.

### 2.1.2 Variable formatting

Variables can be used to prevent hardcoding data into each request, either from included global configuration files or saving data from previous stages of a test (how these variables are ‘injected’ into a test is described in more detail in the relevant sections).

An example of accessing a string from a configuration file which is then passed in the request:

```py
request:
  json:
    variable_key: "{key_name:s}"
# or
# variable_key: "{key_name}"
```

This is formatted using Python’s [string formatting syntax](https://docs.python.org/3/tutorial/inputoutput.html). The variable to be used is encased in curly brackets and an optional **type code** can be passed after a colon.
This means that if you want to pass a literal { or } in a request (or expect it in a response), it must be escaped by doubling it:

```request
  json
  graphql_query: "{%raw%}{ user(id: 123) {{ first_name }} }{%endraw%}"
```

Since 0.5.0, Tavern also has some ‘magic’ variables available in the tavern key for formatting.

**Request variables**

This currently includes all request variables and is available under the request_vars key. Say we want to test a server that updates a user’s profile and returns the change:

```---
  test_name: Check server responds with updated data
  stages:
    - name: Send message, expect it to be echoed back
      request:
        method: POST
        url: "www.example.com/user"
        json:
          welcome_message: "hello"
          params:
            user_id: abc123
      response:
        status_code: 200
        json:
          user_id: "{{tavern.request_vars.params.user_id}}"
          new_welcome_message: "{{tavern.request_vars.json.welcome_message}}"
---
```

This example uses json and params - we can also use any of the other request parameters like method, url, etc.

**Environment variables**

Environment variables are also available under the env_vars key. If a server being tested against requires a password, bearer token, or some other form of authorisation that you don’t want to ship alongside the test code, it can be accessed via this key (for example, in CI).

```---
  test_name: Test getting user information requires auth
  stages:
    - name: Get information without auth fails
      request:
        method: GET
        url: "www.example.com/get_info"
      response:
        status_code: 401
        json:
          error: "No authorization"
    - name: Get information with admin token
      request:
```

(continues on next page)
2.1.3 Calling external functions

Not every response can be validated simply by checking the values of keys, so with Tavern you can call external functions to validate responses and save decoded data. You can write your own functions or use those built in to Tavern. Each function should take the response as its first argument, and you can pass extra arguments using the `extra_kwargs` key.

To make sure that Tavern can find external functions you need to make sure that it is in the Python path. For example, if `utils.py` is in the ‘tests’ folder, you will need to run your tests something like (on Linux):

```bash
$ PYTHONPATH=$PYTHONPATH:tests py.test tests/
```

Checking the response using external functions

The function(s) should be put into the `verify_response_with` block of a response (HTTP or MQTT):

```python
name: Check friendly mess
request:
  url: "{(host)/token"
  method: GET
response:
  status_code: 200
  verify_response_with:
    function: testing_utils:message_says_hello
```

# testing_utils.py

```python
def message_says_hello(response):
    """Make sure that the response was friendly
    ""
    assert response.json().get("message") == "hello world"
```

A list of functions can also be passed to `verify_response_with` if you need to check multiple things:

```python
response:
  status_code: 200
  verify_response_with:
    function: testing_utils:message_says_hello
    function: testing_utils:message_says_something_else
    extra_kwargs:
      should_say: hello
```

**Built-in validators**

There are two external functions built in to Tavern: `validate_jwt` and `validate_pykwalify`.

2.1. Basic Concepts
validate_jwt takes the key of the returned JWT in the body as jwtaas. and additional arguments that are passed directly to the decode method in the PyJWT library. **NOTE:** Make sure the keyword arguments you are passing are correct or PyJWT will silently ignore them. In the future, this function will likely be changed to use a different library to avoid this issue.

```python
# Make sure the response contains a key called 'token', the value of which is a valid jwt which is signed by the given key.

response:
  verify_response_with:
    function: tavern.testutils.helpers:validate_jwt
    extra_kwargs:
      jwt_key: "token"
      key: CGQgaG7GYvTapaQ2qos14
      options:
        verify_signature: true
        verify_aud: false
```

validate_pykwalify takes a pykwalify schema and verifies the body of the response against it.

```python
# Make sure the response matches the given schema - a sequence of dictionaries, which has to contain a user name and may contain a user number.

response:
  verify_response_with:
    function: tavern.testutils.helpers:validate_pykwalify
    extra_kwargs:
      schema:
        type: seq
        required: True
        sequence:
          type: map
          mapping:
            user_number:
              type: int
              required: False
            user_name:
              type: str
              required: True
```

If an external function you are using raises any exception, the test will be considered failed. The return value from these functions is ignored.

### Using external functions for other things

External functions can be used to inject arbitrary data into tests or to save data from the response.

An external function must return a dict where each key either points to a single value or to an object which is accessible using dot notation. The easiest way to do this is to return a Box object.

**Note:** Functions used in the `verify_response_with` block in the `response` block take the response as the first argument. Functions used anywhere else should take *no* arguments. This might be changed in future to be less confusing.

### Injecting external data into a request

A use case for this is trying to insert some data into a response that is either calculated dynamically or fetched from an external source. If we want to generate some authentication headers to access our API for example, we can use an...
external function using the $\texttt{ext}$ key to calculate it dynamically (note as above that this function should \textit{not} take any arguments):

```python
# utils.py
from box import Box

def generate_bearer_token():
    token = sign_a_jwt()
    auth_header = {
        "Authorization": "Bearer ".format(token)
    }
    return Box(auth_header)
```

This can be used as so:

```json
name: login
request:
    url: http://server.com/login
    headers:
        $ext:
            function: utils:generate_bearer_token
    json:
        username: test_user
        password: abc123
response:
    status_code: 200
```

By default, using the $\texttt{ext}$ key will replace anything already present in that block. Input from external functions can be merged into a request instead by specifying the \texttt{tavern-merge-ext-function-values} option in your pytest.ini or on the command line:

```python
# ext_functions.py
def return_hello():
    return "hello": "there"
```

```json
request:
    url: "/(host)/echo"
    method: POST
    json:
        goodbye: "now"
        $ext:
            function: ext_functions:return_hello
```

If \texttt{tavern-merge-ext-function-values} is set, this will send “hello” and “goodbye” in the request. If not, it will just send “hello”.

### Saving data from a response

When using the $\texttt{ext}$ key in the \texttt{save} block there is special behaviour - each key in the returned object will be saved as if it had been specified separately in the \texttt{save} object. The function is called in the same way as a validator function, in the $\texttt{ext}$ key of the \texttt{save} object.

Say that we have a server which returns a response like this:

```json
2.1. Basic Concepts 11
```
If our test function extracts the key `name` from the response body (note as above that this function should take the response object as the first argument):

```python
# utils.py
from box import Box
def test_function(response):
    return Box({'test_user_name': response.json()['user']['name']})
```

We would use it in the `save` object like this:

```yaml
$ext:
    function: utils:test_function
generic:
    test_user_id: user.id
```

In this case, both `{test_user_name}` and `{test_user_id}` are available for use in later requests.

### A more complicated example

For a more practical example, the built-in `validate_jwt` function also returns the decoded token as a dictionary wrapped in a `Box` object, which allows dot-notation access to members. This means that the contents of the token can be used for future requests. Because Tavern will already be in the Python path (because you installed it as a library) you do not need to modify the `PYTHONPATH`.

For example, if our server saves the user ID in the `sub` field of the JWT:

```yaml
name: login
request:
    url: http://server.com/login
    json:
        username: test_user
        password: abc123
response:
    status_code: 200
    verify_response_with:
        # Make sure a token exists
        function: tavern.testutils.helpers:validate_jwt
        extra_kwargs:
            jwt_key: "token"
        options:
            verify_signature: false
    save:
        # Saves a jwt token returned as 'token' in the body as 'jwt'
        # in the test configuration for use in future tests
        # Note the use of $ext again
        $ext:
            function: tavern.testutils.helpers:validate_jwt
```

(continues on next page)
Ideas for other helper functions which might be useful:

- Making sure that the response matches a database schema
- Making sure that an error returns the correct error text in the body
- Decoding base64 data to extract some information for use in a future query
- Validate templated HTML returned from an endpoint using an XML parser
- etc.

One thing to bear in mind is that data can only be saved for use within the same test - each YAML document is considered to be a separate test (not counting anchors as described below). If you need to use the data in multiple tests, you will either need to put it into another file which you then include, or perform the same request in each test to re-fetch the data.

### 2.1.4 Strict key checking

‘Strict’ key checking can be enabled or disabled globally, per test, or per stage. ‘Strict’ key checking refers to whether extra keys in the response should be ignored or whether they should raise an error. With strict key checking enabled, all keys in dictionaries at all levels have to match or it will raise an error. With it disabled, Extra keys in the response will be ignored as long as the ones in your response block are present.

Strict key checking can be controlled individually for the response for the JSON body, the redirect query parameter, or the headers.

By default, strict key checking is disabled for headers and redirect query parameters in the response, but enabled for JSON (as well as when checking for JSON in an mqtt response). This is because although there may be a lot of ‘extra’ things in things like the response headers (such as server agent headers, cache control headers, etc), the expected JSON body will likely always want to be matched exactly.

#### Effect of different settings

This is best explained through an example. If we expect this response from a server:

```
"first": 1
"second":
  "nested": 2
```

This is what we would put in our Tavern test:
The behaviour of various levels of ‘strictness’ based on the response:

Turning ‘strict’ off also means that extra items in lists will be ignored as long as the ones specified in the test response are present. For example, if the response from a server is \([ 1, 2, 3 ]\) then strict being on - the default for the JSON response body - will match only \([1, 2, 3]\).

With strict being turned off for the body, any of these in the test will pass:

- \([1, 2, 3]\)
- \([1]\)
- \([2]\)
- \([3]\)
- \([1, 2]\)
- \([2, 3]\)
- \([1, 3]\)

But not:

- \([3, 1], [2, 1]\) - items present, but out of order
- \([2, 4]\) - ‘4’ not present in response from the server

**Changing the setting**

This setting can be controlled in 3 different ways, the order of priority being:

1. In the test/stage itself
2. Passed on the command line
3. Read from pytest config

This means that using the command line option will not override any settings for specific tests.

Each of these methods is done by passing a sequence of strings indicating which section (json/redirect_query_params=headers) should be affected, and optionally whether it is on or off.

- json:off headers:on - turn off for the body, but on for the headers. redirect_query_params will stay default off.
- json:off headers:off - turn body and header strict checking off
- redirect_query_params:on json:on redirect parameters is turned on and json is kept on (as it is on by default), header strict matching is kept off (as default).

Leaving the ‘on’ or ‘off’ at the end of each setting will imply ‘on’ - ie, using json headers redirect_query_params as an option will turn them all on.
Command line

There is a command line argument, \(--\text{tavern-strict}\), which controls the default global strictness setting.

```
# Enable strict checking for body and headers only
py.test --tavern-strict json:on headers:on redirect_query_params:off -- my_test_folder/
```

In the Pytest config file

This behaves identically to the command line option, but will be read from whichever configuration file Pytest is using.

```
[pytest]
tavern-strict=json:off headers:on
```

Per test

Strictness can also be enabled or disabled on a per-test basis. The `strict` key at the top level of the test should a list consisting of one or more strictness setting as described in the previous section.

```
test_name: Make sure the headers match what I expect exactly
strict
  headers:on
  json:off
stages
  name: Try to get user
  request
    url: "\{host\}/users/joebloggs"
    method: GET
  response
    status_code: 200
    headers:
      content-type: application/json
      content-length: 20
      x-my-custom-header: chocolate
    json
      # As long as "id: 1" is in the response, this will pass and other keys will be ignored
      id: 1
```

A special option that can be done at the test level (or at the stage level, as described in the next section) is just to pass a boolean. This will turn strict checking on or off for all settings for the duration of that test/stage.

```
test_name: Just check for one thing in a big nested dict
# completely disable strict key checking for this whole test
strict: False
stages
  name: Try to get user
```

(continues on next page)
Per stage

Often you have a standard stage before other stages, such as logging in to your server, where you only care if it returns a 200 to indicate that you’re logged in. To facilitate this, you can enable or disable strict key checking on a per-stage basis as well.

Two examples for doing this - these examples should behave identically:

```json
# Enable strict checking for this test, but disable it for the login stage

# Force re-enable strict checking, in case it was turned off globally

stages:
  - name: log in
    request:
      url: "{host}/users/joebloggs"
      method: GET
    response:
      # Disable all strict key checking just for this stage
      strict: False
      status_code: 200
      json:
        logged_in: True
        # Ignores any extra metadata like user id, last login, etc.

  - name: Create a new user
    request:
      url: "{host}/users/joebloggs"
      method: POST
    response:
      status_code: 200
      # Because strict was set 'on' at the test level, this must match exactly
      json:
        <<: *create_user
```

(continues on next page)
Or if strict json key checking was enabled at the global level:

```
test_name: Login and create a new user

stages:
  - name: log in
    request:
      url: "{host}/users/joebloggs"
      method: GET
    response:
      strict:
        json: off
        status_code: 200
        json:
          logged_in: True
  - name: Create a new user
    request: ...
```

### 2.1.5 Reusing requests and YAML fragments

A lot of tests will require using the same step multiple times, such as logging in to a server before running tests or simply running the same request twice in a row to make sure the same (or a different) response is returned.

Anchors are a feature of YAML which allows you to reuse parts of the code. Define an anchor using `&name_of_anchor`. This can then be assigned to another object using `new_object: *name_or_anchor`, or they can be used to extend objects using `<<: *name_of_anchor`.

```yaml
# input.yaml

first: &top_anchor
  a: b
  c: d

second: *top_anchor

third: 
  <<: *top_anchor
  c: overwritten
  e: f
```

If we convert this to JSON, for example with a script like this:

```python
#!/usr/bin/env python

# load.py
import yaml
import json

with open("input.yaml", "r") as yfile:
```

(continues on next page)
for doc in yaml.load_all(yfile.read()):
    print(json.dumps(doc, indent=2))

We get something like the following:

```
'first': {
  'a': 'b',
  'c': 'd'
},
'second': {
  'a': 'b',
  'c': 'd'
},
'third': {
  'a': 'b',
  'c': 'overwritten',
  'e': 'f'
}
```

This does not however work if there are different documents in the yaml file:

```
# input.yaml
first: &top_anchor
  a: b
  c: d
second: *top_anchor
---
third:
<<: *top_anchor
c: overwritten
e: f
```

```
$ python test.py
{
  "second": {
    "c": "d",
    "a": "b"
  },
  "first": {
    "c": "d",
    "a": "b"
  }
}
```

Traceback (most recent call last):
  File "test.py", line 8, in <module>
    for doc in yaml.load_all(yfile.read()):
    File "/home/cooldeveloper/.virtualenvs/tavern/lib/python3.5/site-packages/yaml/__init__.py", line 84, in load_all
      yield loader.get_data()
    File "/home/cooldeveloper/.virtualenvs/tavern/lib/python3.5/site-packages/yaml/constructor.py", line 31, in get_data
```
This poses a bit of a problem for running our integration tests. If we want to log in at the beginning of each test, or if we want to query some user information which is then operated on for each test, we don’t want to copy paste the same code within the same file.

For this reason, Tavern will override the default YAML behaviour and preserve anchors across documents within the same file. Then we can do something more like this:

```
---
test_name: Make sure user location is correct
stages:
- &test_user_login_anchor
  # Log in as user and save the login token for future requests
  name: Login as test user
  request:
    url: http://test.server.com/user/login
    method: GET
    json:
      username: test_user
      password: abc123
  response:
    status_code: 200
    save:
      json:
        test_user_login_token: token
    verify_response_with:
      function: tavern.testutils.helpers:validate_jwt
      extra_kwargs:
        jwt_key: "token"
      options:
        verify_signature: false
```

(continues on next page)
- name: Get user location
  request:
    url: http://test.server.com/locations
    method: GET
    headers:
      Authorization: "Bearer {test_user_login_token}"
  response
  status_code: 200
  json
  location
    road: 123 Fake Street
    country: England

---

- name: Make sure giving premium works
  stages:
    # Use the same block to log in across documents
    - *test_user_login_anchor

      - name: Assert user does not have premium
        request: &has_premium_request_anchor
          url: http://test.server.com/user_info
          method: GET
          headers:
            Authorization: "Bearer {test_user_login_token}"
        response
          status_code: 200
          json:
            has_premium: false

      - name: Give user premium
        request:
          url: http://test.server.com/premium
          method: POST
          headers:
            Authorization: "Bearer {test_user_login_token}"
        response
          status_code: 200

      - name: Assert user now has premium
        request: <<: *has_premium_request_anchor
        response
          status_code: 200
          json:
            has_premium: true

2.1.6 Including external files

Even with being able to use anchors within the same file, there is often some data which either you want to keep in a separate (possibly autogenerated) file, or is used on every test (e.g. login information). You might also want to run the same tests with different sets of input data.
Because of this, external files can also be included which contain simple key: value data to be used in other tests.

Including a file in every test can be done by using a `!include` directive:

```yaml
# includes.yaml

# Each file should have a name and description
name: Common test information
description: Login information for test server

# Variables should just be a mapping of key: value pairs
variables:
  protocol: https
  host: www.server.com
  port: 1234

# tests.tavern.yaml

---

test_name: Check server is up
includes:
  - !include includes.yaml

stages:
  - name: Check healthz endpoint
    request:
      method: GET
      url: '{protocol:s}://{host:s}:{port:d}'
    response:
      status_code: 200
```

As long as includes.yaml is in the same folder as the tests or found in the TAVERN_INCLUDE search path, the variables will automatically be loaded and available for formatting as before. Multiple include files can be specified.

The environment variable TAVERN_INCLUDE can contain a : separated list of paths to search for include files. Each path in TAVERN_INCLUDE has environment variables expanded before it is searched.

### Including global configuration files

If you do want to run the same tests with a different input data, this can be achieved by passing in a global configuration. Using a global configuration file works the same as implicitly including a file in every test. For example, say we have a server that takes a user’s name and address and returns some hash based on this information. We have two servers that need to do this correctly, so we need two tests that use the same input data but need to post to 2 different urls:

```yaml
# two_tests.tavern.yaml

---

test_name: Check server A responds properly
includes:
  - !include includesA.yaml

stages:
  - name: Check thing is processed correctly
    request:
      method: GET
```

(continues on next page)
Including the full set of input data in includesA.yaml and includesB.yaml would mean that a lot of the same input data would be repeated. To get around this, we can define a file called, for example, common.yaml which has all the input data except for host in it, and make sure that includesA/B only have the host variable in:

```yaml
# common.yaml

name: Common test information
description: user location information for Joe Bloggs test user

variables:
  name: Joe Bloggs
  house_number: 123
  street: Fake street
  town: Chipping Sodbury
  postcode: BS1 2BC
  country: England
  planet: Earth
  galaxy: Milky Way
  universe: A
  expected_hash: aJdaAK4f5x5Wzxtr8WtkLC5
```

*Tavern Documentation, Release 1.12.2*

---

name: server A information
description: server A specific information
variables:
  - host: www.server-a.com

---

name: server B information
description: server B specific information
variables:
  - host: www.server-B.io

If the behaviour of server A and server B ever diverge in future, information can be moved out of the common file and into the server specific include files.

Using the **tavern-ci** tool or **pytest**, this global configuration can be passed in at the command line using the `--tavern-global-cfg` flag. The variables in `common.yaml` will then be available for formatting in **all** tests during that test run.

**NOTE:** **tavern-ci** is just an alias for **py.test** and will take the same options.

```
$ tavern-ci --tavern-global-cfg=integration_tests/local_urls.yaml
$ tavern-ci --tavern-global-cfg integration_tests/local_urls.yaml
$ py.test --tavern-global-cfg=integration_tests/local_urls.yaml
$ py.test --tavern-global-cfg integration_tests/local_urls.yaml
```

It might be tempting to put this in the ‘addopts’ section of the `pytest.ini` file to always pass a global configuration when using **pytest**, but be careful when doing this - due to what appears to be a bug in the **pytest** option parsing, this might not work as expected:

```
# py.test.ini
[pytest]
addopts =
  # This will work
  --tavern-global-cfg=integration_tests/local_urls.yaml
  # This will not!
  # --tavern-global-cfg integration_tests/local_urls.yaml
```

Instead, use the `tavern-global-cfg` option in your `pytest.ini` file:

```
[pytest]
tavern-global-cfg=
  integration_tests/local_urls.yaml
```

**Multiple global configuration files**

Sometimes you will want to have 2 (or more) different global configuration files, one containing common information such as paths to different resources and another containing information specific to the environment that is being tested.
Multiple global configuration files can be specified either on the command line or in pytest.ini to avoid having to put an `!include` directive in every test:

```bash
$ tavern-ci --tavern-global-cfg common.yaml test_urls.yaml -- test_server.tavern.yaml
$ py.test --tavern-global-cfg common.yaml local_docker_urls.yaml -- test_server.→tavern.yaml
```

Sharing stages in configuration files

If you have a stage that is shared across a huge number of tests and it is infeasible to put all the tests which share that stage into one file, you can also define stages in configuration files and use them in your tests.

Say we have a login stage that needs to be run before every test in our test suite. Stages are defined in a configuration file like this:

```yaml
# auth_stage.yaml

name: Authentication stage
description: Reusable test stage for authentication
variables:
  user:
    user: test-user
    pass: correct-password
stages:
  id: login_get_token
  name: Login and acquire token
  request:
    url: "%(service:s)/login"
    json:
      user: "%(user.user:s)"
      password: "%(user.pass:s)"
    method: POST
    headers:
      content-type: application/json
  response:
    status_code: 200
    headers:
      content-type: application/json
    save:
      json:
        test_login_token: token
```

Each stage should have a uniquely identifiable `id`, but other than that the stage can be define just as other tests (including using format variables).
This can be included in a test by specifying the `id` of the test like this:

```yaml
---
test_name: Test authenticated /hello
includes:
  - !include auth_stage.yaml
stages:
  - type: ref
    id: login_get_token
    name: Authenticated /hello
    request:
      url: "{(service:s)}/hello/Jim"
      method: GET
      headers:
        Content-Type: application/json
        Authorization: "Bearer {test_login_token}"
    response:
      status_code: 200
      headers:
        content-type: application/json
        json:
          data: "Hello, Jim"
```

### Directly including test data

If your test just has a huge amount of data that you would like to keep in a separate file, you can also (ab)use the `!include` tag to directly include data into a test. Say we have a huge amount of JSON that we want to send to a server and we don’t want hundreds of lines in the test:

```json
// test_data.json
[
  {
    "_id": "5c965b1373f3fe071a9cb2b7",
    "index": 0,
    "guid": "ef3f8c42-522a-4d6b-84ec-79a07009460d",
    "isActive": false,
    "balance": "$3,103.47",
    "picture": "http://placehold.it/32x32",
    "age": 26,
    "eyeColor": "green",
    "name": "Cannon Wood",
    "gender": "male",
    "company": "CANDECOR",
    "email": "cannonwood@candecor.com",
    "phone": "+1 (944) 549-2826",
    "address": "528 Woodpoint Road, Snowville, Kansas, 140",
    "about": "Dolore in consequat exercitation esse esse velit eu velit aliquip ex.\n\nReprehenderit est consectetur excepteur sint sint dolore. Anim minim dolore est ut\n\nfugiat. Occaecat tempor tempor mollit dolore anim commodo laboris commodo aute quis\n\nex irure voluptate. Sunt magna tempor veniam cillum exercitation quis minim est\n\neiusmod aliqua.\n\n"",
    "registered": "2015-12-27T11:30:18 -00:00",
    "latitude": -2.515302,
```

(continues on next page)
"longitude": -98.678105,
"tags": [
  "proident",
  "aliqua",
  "velit",
  "labore",
  "consequat",
  "esse",
  "ea"
],
"friends": [
  {
  "id": 0,
  "etc": []
  }
]
}

(Handily generated by JSON Generator)

Putting this whole thing into the test would be a bit overkill, but it can be inject directly into your test like this:

```yaml
---
test_name: Post a lot of data
stages:
  - name: Create new user
    request:
      url: "{service:s}/new_user"
      method: POST
      json: !include test_data.json
    response:
      status_code: 201
      json:
        status: user created
```

This works with YAML as well, the only caveat being that the filename must end with .yaml, .yml, or .json.

### 2.1.7 Using the run() function

Because the `run()` function (see examples) calls directly into the library, there is no nice way to control which global configuration to use - for this reason, you can pass a dictionary into `run()` which will then be used as global configuration. This should have the same structure as any other global configuration file:

```python
from tavern.core import run

extra_cfg = {
    "variables": {
        "key_1": "value",
        "key_2": 123
    }
}

success = run("test_server.tavern.yaml", extra_cfg)
```
An absolute filepath to a configuration file can also be passed.

This is also how things such as strict key checking is controlled via the `run()` function. Extra keyword arguments that are taken by this function:

- `tavern_strict` - Controls strict key checking (see section on strict key checking for details)
- `tavern_mqtt_backend` and `tavern_http_backend` controls which backend to use for those requests (see plugins for details)
- `pytest_args` - A list of any extra arguments you want to pass directly through to Pytest.

An example of using `pytest_args` to exit on the first failure:

```python
from tavern.core import run
success = run("test_server.tavern.yaml", pytest_args=["-x"])
```

`run()` will use a Pytest instance to actually run the tests, so these values can also be controlled just by putting them in the appropriate Pytest configuration file (such as your `setup.cfg` or `pytest.ini`).

Under the hood, the `run` function calls `pytest.main` to start the test run, and will pass the return code back to the caller. At the time of writing, this means it will return a `0` if all tests are successful, and a nonzero result if one or more tests failed (or there was some other error while running or collecting the tests).

### 2.1.8 Matching arbitrary return values in a response

Sometimes you want to just make sure that a value is returned, but you don’t know (or care) what it is. This can be achieved by using `!anything` as the value to match in the `response` block:

```yaml
response:
  json:
    # Will assert that there is a 'returned_uuid' key, but will do no checking
    # on the actual value of it
    returned_block: !anything
```

This would match both of these response bodies:

```yaml
returned_block: hello
```

```yaml
returned_block:
  nested: value
```

Using the magic `!anything` value should only ever be used inside pre-defined blocks in the response block (for example, `headers`, `params`, and `json` for a HTTP response).

**NOTE**: Up until version 0.7.0 this was done by setting the value as `null`. This creates issues if you want to ensure that your server is actually returning a null value. Using `null` is still supported in the current version of Tavern, but will be removed in a future release, and should raise a warning.

### Matching arbitrary specific types in a response

If you want to make sure that the key returned is of a specific type, you can use one of the following markers instead:

- `!anyint`: Matches any integer
- `!anyfloat`: Matches any float (note that this will NOT match integers!)
- `!anystr`: Matches any string
• !anybool: Matches any boolean (this will NOT match null)
• !anylist: Matches any list
• !anydict: Matches any dict/'mapping'

Matching via a regular expression

Sometimes you know something will be a string, but you also want to make sure that the string matches some kind of regular expression. This can be done using external functions, but as a shorthand there is also the !re_ family of custom YAML tags that can be used to match part of a response. Say that we want to make sure that a UUID returned is a version 4 UUID, where the third block must start with 4 and the third block must start with 8, 9, “A”, or “B”.

```yaml
name: Check that uuidv4 is returned
request:
  url: {host}/get_uuid/v4
  method: GET
response:
  status_code: 200
  json:
    uuid: !re_fullmatch "[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89AB][0-9a-f]{3}-
      [0-9a-f]{12}"
```

This is using the !re_fullmatch variant of the tag - this calls re.fullmatch under the hood, which means that the regex given needs to match the entire part of the response that is being checked for it to pass. There is also !re_search which will pass if it matches part of the thing being checked, or !re_match which will match part of the thing being checked, as long as it is at the beginning of the string. See the Python documentation for more details.

2.1.9 Type conversions

YAML has some magic variables that you can use to coerce variables to certain types. For example, if we want to write an integer but make sure it gets converted to a string when it’s actually sent to the server we can do something like this:

```yaml
request:
  json:
    an_integer: !!str 1234567890
```

However, due to the way YAML is loaded this doesn’t work when you are using a formatted value. Because of this, Tavern provides similar special constructors that begin with a single exclamation mark that will work with formatted values. Say we want to convert a value from an included file to an integer:

```yaml
request:
  json:
    # an_integer: !!int "{my_integer:d}" # Error
    an_integer: !int "{my_integer:d}" # Works
```

Because curly braces are automatically formatted, trying to send one in a string might cause some unexpected issues. This can be mitigated by using the !raw tag, which will not perform string formatting.

Note: This is just shorthand for replacing a `{ with a {{ in the string
Including raw JSON data

Sometimes there are situations where you need to directly include a block of JSON, such as a list, rather than just one value. To do this, there is a `!force_format_include` tag which will include whatever variable is being referenced in the format block rather than coercing it to a string.

For example, if we have an API that will return a list of users on a GET and will bulk delete a list of users on a DELETE, a test that all users are deleted could be done by

1. GET all users
2. DELETE the list you just got
3. GET again and expect an empty list

```
- name: Get all users
  request:
    url: "/{host}/users"
    method: GET
  response:
    status_code: 200
    # Expect a list of users
    json: !anylist
  save:
    json:
      # Save the list as 'all_users'
      all_users: "@"

- name: delete all users
  request:
    url: "/{host}/users"
    method: DELETE
    # 'all_users' list will be sent in the request as a list, not a string
    json: !force_format_include "{all_users}"
  response:
    status_code: 204

- name: Get no users
  request:
    url: "/{host}/users"
    method: GET
  response:
    status_code: 200
    # Expect no users
    json: []
```

Any blocks of JSON that are included this way will not be recursively formatted. When using this token, do not use a conversion specifier (eg `{all_users:s}`) as it will be ignored.

### 2.1.10 Adding a delay between tests

Sometimes you might need to wait for some kind of uncontrollable external event before moving on to the next stage of the test. To wait for a certain amount of time before or after a test, the `delay_before` and `delay_after` keys
can be used. Say you have an asynchronous task running after sending a POST message with a user id - an example of using this behaviour:

```yaml
---

test_name: Make sure asynchronous task updates database
stages:
  - name: Trigger task
    request:
      url: https://example.com/run_intensive_task_in_background
      method: POST
      json:
        user_id: 123
      # Server responds instantly...
      response:
        status_code: 200
      # ...but the task takes ~3 seconds to complete
      delay_after: 5
    
  - name: Check task has triggered
    request:
      url: https://example.com/check_task_triggered
      method: POST
      json:
        user_id: 123
      response:
        status_code: 200
        json:
          task: completed
```

Having `delay_before` in the second stage of the test is semantically identical to having `delay_after` in the first stage of the test - feel free to use whichever seems most appropriate.

A saved/config variable can be used by using a type token conversion, such as:

```yaml
stages:
  - name: Trigger task
    ...
  delay_after: !float "{sleep_time}"```

### 2.1.11 Retrying tests

If you are not sure how long the server might take to process a request, you can also retry a stage a certain number of times using `max_retries`:

```yaml
test_name: Poll until server is ready
includes:
  - !include common.yaml
stages:
  - name: polling
    max_retries: 1
    request:
      url: "{(host)/poll"
```

(continues on next page)
This example will perform a GET request against /poll, and if it does not return the expected response, will try one more time, immediately. To wait before retrying a request, combine max_retries with delay_after.

NOTE: You should think carefully about using retries when making a request that will change some state on the server or else you may get nondeterministic test results.

MQTT tests can be retried as well, but you should think whether this is what you want - you could also try increasing the timeout on an expected MQTT response to achieve something similar.

### 2.1.12 Marking tests

Since 0.11.0, it is possible to 'mark' tests. This uses Pytest behind the scenes - see the pytest mark documentation for details on their implementation and prerequisites for use.

In short, marks can be used to:

- Select a subset of marked tests to run from the command line
- Skip certain tests based on a condition
- Mark tests as temporarily expected to fail, so they can be fixed later

An example of how these can be used:

```python
# test_name: Get server info from slow endpoint
marks:
  - slow
stages:
  - name: Get info
    request:
      url: "{host}/get-info-slow"
      method: GET
      response:
        status_code: 200
        json:
          n_users: 2048
          n_queries: 10000

# test_name: Get server info from fast endpoint
marks:
  - fast
stages:
  - name: Get info
    request:
      url: "{host}/get-info"
```

(continues on next page)
Both tests get some server information from our endpoint, but one requires a lot of backend processing so we don’t want to run it on every test run. This can be selected like this:

```
$ py.test -m "not slow"
```

Conversely, if we just want to run all tests marked as ‘fast’, we can do this:

```
$ py.test -m "fast"
```

Marks can only be applied to a whole test, not to individual stages (with the exception of `skip`, see below).

### Formatting marks

Marks can be formatted just like other variables:

```
---

test_name: Get server info from slow endpoint

marks:
  - "{specialmarker}"
```

This is mainly for combining with one or more of the special marks as mentioned below.

**NOTE:** Do not use the `!raw` token or rely on double curly brace formatting when formatting markers. Due to `pytest-xdist`, some behaviour with the formatting of markers is subtly different than other places in Tavern.

### Special marks

There are 4 different ‘special’ marks from Pytest which behave the same as if they were used on a Python test.

**NOTE:** If you look in the Tavern integration tests, you may notice a `_xfail` key being used in some of the tests. This is for INTERNAL USE ONLY and may be removed in future without warning.

#### skip

To always skip a test, just use the `skip` marker:

```
---

marks:
  - skip
```

Separately from the markers, individual stages can be skipped by inserting the `skip` keyword into the stage:
Sometimes you just want to skip some tests, perhaps based on which server you’re using. Taking the above example of the ‘slow’ server, perhaps it is only slow when running against the live server at www.slow-example.com, but we still want to run it in our local tests. This can be achieved using **skipif**:  

```yaml
---
test_name: Get server info from slow endpoint
marks:
  - slow
  - skipif: "'slow-example.com' in '{host}'"
stages:
  - name: Get info
    request:
      url: "{(host)}/get-info-slow"
      method: GET
      response:
        status_code: 200
        json:
          n_users: 2048
          n_queries: 10000
```

**skipif** should be a mapping containing 1 key, a string that will be directly passed through to `eval()` and should return `True` or `False`. This string will be formatted first, so tests can be skipped or not based on values in the configuration. Because this needs to be a valid piece of Python code, formatted strings must be escaped as in the example above - using "'slow-example.com' in '{host}'" will raise an error.

**xfail**

If you are expecting a test to fail for some reason, such as if it’s temporarily broken, a test can be marked as **xfail**. Note that this is probably not what you want to ‘negatively’ check something like an API deprecation. For example, this is not recommended:

```yaml
---
test_name: Get user middle name from endpoint on v1 api
stages:
  - name: Get from endpoint
    request:
```

(continues on next page)
It would be much better to write a test that made sure that the endpoint just returned a 404 in the v2 api.

**parametrize**

A lot of the time you want to make sure that your API will behave properly for a number of given inputs. This is where the parametrize mark comes in:

```
    test_name: Make sure backend can handle arbitrary data

    marks
    parametrize
    key: metadata
    vals
        13:00
        Reading: 27 degrees
        
    stages
    name: Update metadata
    request:
        url: "{(host)/devices/{device_id}/metadata"}
        method: POST
        json:
            metadata: "{(metadata)}"
        response
        status_code: 200
```

This test will be run 4 times, as 4 separate tests, with metadata being formatted differently for each time. This behaves like the built in Pytest parametrize mark, where the tests will show up in the log with some extra data appended to show what was being run, eg Test Name[John], Test Name[John-Smythe John], etc.
The `parametrize` mark should be a mapping with `key` being the value that will be formatted and `vals` being a list of values to be formatted. Note that formatting of these values happens after checking for a `skipif`, so a `skipif` mark cannot rely on a parametrized value.

Multiple marks can be used to parametrize multiple values:

```json
---
test_name: Test post a new fruit
marks:
  - parametrize:
      key: fruit
      vals:
        - apple
        - orange
        - pear
  - parametrize:
      key: edible
      vals:
        - rotten
        - fresh
        - unripe
stages:
  - name: Create a new fruit entry
    request:
      url: "{host}/fruit"
      method: POST
      json:
        fruit_type: "{edible} {fruit}"
    response:
      status_code: 201
---
```

This will result in 9 tests being run:

- rotten apple
- rotten orange
- rotten pear
- fresh apple
- fresh orange
- etc.

If you need to parametrize multiple keys but don’t want there to be a new test created for every possible combination, pass a list to `key` instead. Each item in `val` must then also be a list that is the same length as the `key` variable. Using the above example, perhaps we just want to test the server works correctly with the items “rotten apple”, “fresh orange”, and “unripe pear” rather than the 9 combinations listed above. This can be done like this:

```json
---
test_name: Test post a new fruit
marks:
  - parametrize:
      key:
        - fruit
        - edible
---
```
vals:
- [rotten, apple]
- [fresh, orange]
- [unripe, pear]

# NOTE: we can specify a nested list like this as well:
# -
#  - unripe
#  - pear

stages:
  name: Create a new fruit entry
  request:
    url: "{host}/fruit"
    method: POST
    json:
      fruit_type: "{edible} {fruit}"
  response:
    status_code: 201

This will result in only those 3 tests being generated.

This can be combined with the ‘simpler’ style of parametrisation as well - for example, to run the above test but also to specify whether the fruit was expensive or cheap:

test_name: Test post a new fruit and price

marks:
  parametrize:
    key:
      fruit
      edible
    vals:
      - [rotten, apple]
      - [fresh, orange]
      - [unripe, pear]
  parametrize:
    key: price
    vals:
      - expensive
      - cheap

stages:
  name: Create a new fruit entry
  request:
    url: "{host}/fruit"
    method: POST
    json:
      fruit_type: "{price} {edible} {fruit}"
  response:
    status_code: 201

This will result in 6 tests:

- expensive rotten apple
- expensive fresh orange
- expensive unripe pear
• cheap rotten apple
• cheap fresh orange
• cheap unripe pear

**NOTE:** Due to implementation reasons it is currently impossible to parametrize either the HTTP method or the MQTT QoS parameter.

## usefixtures

Since 0.15.0 there is limited support for Pytest fixtures in Tavern tests. This is done by using the `usefixtures` mark. The return (or `yield`ed) values of any fixtures will be available to use in formatting, using the name of the fixture.

An example of how this can be used in a test:

```python
# conftest.py
import pytest
import logging
import time

@pytest.fixture
def server_password():
    with open("/path/to/password/file", "r") as pfile:
        password = pfile.read().strip()
    return password

@pytest.fixture(name="time_request")
def fix_time_request():
    t0 = time.time()
    yield
    t1 = time.time()
    logging.info("Test took %s seconds", t1 - t0)

test_name: Make sure server can handle a big query
marks:
    usefixtures:
        - time_request
        - server_password
stages:
    name: Do big query
    request:
        url: "/{host}/users"
        method: GET
        params:
            n_items: 1000
        headers:
            authorization: "Basic {server_password}"```
The above example will load basic auth credentials from a file, which will be used to authenticate against the server. It will also time how long the test took and log it.

usefixtures expects a list of fixture names which are then loaded by Pytest - look at their documentation to see how discovery etc. works.

There are some limitations on fixtures:

- Fixtures are per test, not per stage. The above example of timing a test will include the (small) overhead of doing validation on the responses, setting up the requests session, etc. If the test consists of more than one stage, it will time how long both stages took.
- Fixtures should be ‘function’ or ‘session’ scoped. ‘module’ scoped fixtures will raise an error and ‘class’ scoped fixtures may not behave as you expect.
- Parametrizing fixtures does not work - this is a limitation in Pytest.

Fixtures which are specified as autouse can also be used without explicitly using usefixtures in a test. This is a good way to essentially precompute a format variable without also having to use an external function or specify a usefixtures block in every test where you need it.

To do this, just pass the autouse=True parameter to your fixtures along with the relevant scope. Using ‘session’ will evaluate the fixture once at the beginning of your test run and reuse the return value everywhere else it is used:

```python
@pytest.fixture(scope="session", autouse=True)
def a_thing():
    return "abc"
```

test_name: Test autouse fixture

- name: do something with fixture value
  request:
    url: "\{host\}/echo"
    method: POST
    json:
      value: "\{a_thing\}"

2.1.13 Hooks

As well as fixtures as mentioned in the previous section, since version 0.28.0 there is a couple of hooks which can be used to extract more information from tests.

These hooks are used by defining a function with the name of the hook in your conftest.py that take the same arguments with the same names - these hooks will then be picked up at runtime and called appropriately.

**NOTE:** These hooks should be considered a ‘beta’ feature, they are ready to use but the names and arguments they take should be considered unstable and may change in a future release (and more may also be added).

More documentation for these can be found in the docstrings for the hooks in the tavern/testutils/pytesthook/newhooks.py file.
Before every test run

This hook is called after fixtures, global configuration, and plugins have been loaded, but before formatting is done on the test and the schema of the test is checked. This can be used to ‘inject’ extra things into the test before it is run, such as configurations blocks for a plugin, or just for some kind of logging.

Example usage:

```python
import logging

def pytest_tavern_beta_before_every_test_run(test_dict, variables):
    logging.info("Starting test %s", test_dict["test_name"])

    variables["extra_var"] = "abc123"
```

After every response

This hook is called after every response for each stage - this includes HTTP responses, but also MQTT responses if you are using MQTT. This means if you are using MQTT it might be called multiple times for each stage!

Example usage:

```python
def pytest_tavern_beta_after_every_response(expected, response):
    with open("logfile.txt", "a") as logfile:
        logfile.write("Got response: {}".format(response.json()))
```

Before every request

This hook is called just before each request with the arguments passed to the request “function”. By default, this is Session.request (from requests) for HTTP and Client.publish (from paho-mqtt) for MQTT.

Example usage:

```python
import logging

def pytest_tavern_beta_before_every_request(request_args):
    logging.info("Making request: %s", request_args)
```

2.2 HTTP integration testing

The things specified in this section are only applicable if you are using Tavern to test a HTTP API (ie, unless you are specifically checking MQTT or some other plugin).

2.2.1 Using multiple status codes

If the server you are contacting might return one of a few different status codes depending on it’s internal state, you can write a test that has a list of status codes in the expected response.

Say for example we want to try and get a user’s details from a server - if it exists, it returns a 200. If not, it returns a 404. We don’t care which one, as long as it it only one of those two codes.
test_name: Make sure that the server will either return a 200 or a 404

stages:
  - name: Try to get user
    request:
      url: "{host}/users/joebloggs"
      method: GET
    response:
      status_code:
        - 200
        - 404

Note that there is no way to do something like this for the body of the response, so unless you are expecting the same response body for every possible status code, the json key should be left blank.

2.2.2 Sending form encoded data

Though Tavern can only currently verify JSON data in the response, data can be sent using x-www-form-urlencoded encoding by using the data key instead of json in a request. An example of sending form data rather than json:

```
request
  url: "{test_host}/form_data"
  method: POST
  data
    id: abc123
```

2.2.3 Authorisation

Persistent cookies

Tavern uses requests under the hood, and uses a persistent Session for each test. This means that cookies are propagated forward to further stages of a test. Cookies can also be required to pass a test. For example, say we have a server that returns a cookie which then needs to be used for future requests:

```
test_name: Make sure cookie is required to log in

includes
  - !include common.yaml

stages:
  - name: Try to check user info without login information
    request:
      url: "{host}/userinfo"
      method: GET
    response:
      status_code: 401
      json
        error: "no login information"
```

(continues on next page)
This test ensures that a cookie called `session-cookie` is returned from the 'login' stage, and this cookie will be sent with all future stages of that test.

**Choosing cookies**

If you have multiple cookies for a domain, the `cookies` key can also be used in the request block to specify which one to send:
- name: Only send one cookie
  request:
    url: "{host}/expect_cookie"
    method: GET
    cookies:
      - tavern-cookie-1
  response
  status_code: 200
  json
  status: ok

Trying to specify a cookie which does not exist will fail the stage.

To send no cookies, simply use an empty array:

---

test_name: Test receiving and sending cookie

includes
  - !include common.yaml

stages
  - name: get cookie for domain
    request
      url: "{host}/get_cookie"
      method: POST
    response
      status_code: 200
      cookies
        - tavern-cookie-1
  - name: Send no cookies
    request
      url: "{host}/expect_cookie"
      method: GET
      cookies: []
    response
      status_code: 403
      json
      status: access denied

**Overriding cookies**

If you want to override the value of a cookie, then instead of passing a string to the cookies block in the request, use a mapping of cookie name: cookie value:

---

- name: Override cookie value
  request
    url: "{host}/expect_cookie"
    method: GET
    cookies:
      - tavern-cookie-2: abc
  response
  (continues on next page)
status_code: 200
json:
  status: ok

This will create a new cookie with the name `tavern-cookie-2` with the value `abc` and send it in the request. If this cookie already exists from a previous stage, it will be overwritten. Trying to override the cookie multiple times in one stage will cause an error to occur at runtime.

**HTTP Basic Auth**

For a server that expects HTTP Basic Auth, the `auth` keyword can be used in the request block. This expects a list of two items - the first item is the user name, and the second name is the password:

```yaml
---
test_name: Check we can access API with HTTP basic auth
includes:
  - !include common.yaml
stages:
  - name: Get user info
    request:
      url: "{host}/userinfo"
      method: GET
      auth:
        - user@api.com
        - password123
    response:
      status_code: 200
      json:
        user_id: 123
      headers:
        content-type: application/json
```

**Custom auth header**

If you’re using a form of authorisation not covered by the above two examples to authorise against your test server (for example, a JWT-based system), specify a custom `Authorization` header. If you are using a JWT, you can use the built-in `validate_jwt` external function as defined above to check that the claims are what you’d expect.

```yaml
---
test_name: Check we can login then use a JWT to access the API
includes:
  - !include common.yaml
stages:
  - name: login
    request:
      url: "{host}/login"
      json:
```

(continues on next page)
2.2.4 Controlling secure access

Running against an unverified server

If you’re testing against a server which has SSL certificates that fail validation (for example, testing against a local development server with self-signed certificates), the **verify** keyword can be used in the **request** stage to disable certificate checking for that request.

Using self signed certificates

In case you need to use a self-signed certificate to connect to a server, you can use the **cert** key in the request to control which certificates will be used by Requests.

If you just want to pass your client certificate with a request, pass the path to it using the **cert** key:
If you need to pass a SSL key file as well, pass a list of length two with the first element being the certificate and the second being the path to the key:

```
---
test_name: Access an API which requires a client certificate
stages:
  - name: Get user info
    request:
      url: "{host}/userinfo"
      method: GET
      cert: "/path/to/certificate"
      # Or use a format variable:
      # cert: "{cert_path}"
    response
    ...
```

See the Requests documentation for more details about this option.

### 2.2.5 Uploading files as part of the request

To upload a file along with the request, the `files` key can be used:

```
---
test_name: Test files can be uploaded with tavern
includes:
  - !include common.yaml
stages:
  - name: Upload multiple files
    request:
      url: "{host}/fake_upload_file"
      method: POST
    files:
      test_files: "test_files.tavern.yaml"
      common: "common.yaml"
    response
      status_code: 200
```

This expects a mapping of the ‘name’ of the file in the request to the path on your computer.
By default, the sending of files is handled by the Requests library - to see the implementation details, see their documentation.

**Uploading a file as the body of a request**

In some cases it may be required to upload the entire contents of a file in the request body - for example, when posting a binary data blob from a file. This can be done for JSON and YAML using the `!include` tag, but for other data formats the `file_body` key can be used:

```yaml
- name: Upload a file in the request body
  request:
    url: "{host}/data_blob"
    method: POST
    file_body: "/path/to/blobfile"
  response:
    status_code: 200
```

Like the `files` key, this is mutually exclusive with the `json` key.

**Specifying custom content type and encoding**

If you need to use a custom file type and/or encoding when uploading the file, there is a ‘long form’ specification for uploading files. Instead of just passing the path to the file to upload, use the `file_path` and `content_type/content_encoding` in the block for the file:

```yaml
---
test_name: Test files can be uploaded with tavern
stages:
  - name: Upload multiple files
    request:
      url: "{host}/fake_upload_file"
      method: POST
      files:
        # simple style - guess the content type and encoding
        test_files: "test_files.tavern.yaml"
        # long style - specify them manually
        common:
          file_path: "common.yaml"
          content_type: "application/customtype"
          content_encoding: "UTF16"
```

**2.2.6 Timeout on requests**

If you want to specify a timeout for a request, this can be done using the `timeout` parameter:

```yaml
---
test_name: Get server info from slow endpoint
stages:
  - name: Get info
    request:
      url: "{host}/get-info-slow"
```

(continues on next page)
If this request takes longer than 0.5 seconds to respond, the test will be considered as failed. A 2-tuple can also be passed - the first value will be a connection timeout, and the second value will be the response timeout. By default this uses the Requests implementation of timeouts - see their documentation for more details.

### 2.2.7 Redirects

By default, Tavern will not follow redirects. This allows you to check whether an endpoint is indeed redirecting a user to a certain page.

To disable this behaviour, use either the `--tavern-always-follow-redirects` command line flag or set `tavern-always-follow-redirects` to True in your Pytest settings file.

This can also be disabled or enabled on a per-stage basis by using the `follow_redirects` flag:

```yaml
---

test_name: Expect a redirect when setting the flag

stages:
  - name: Expect to be redirected
    request:
      url: "{host}/redirect/source"
      follow_redirects: true
    response:
      status_code: 200
      json:
        status: successful redirect
```

Specifying `follow_redirects` on a stage will override any global setting, so if you just want to change the behaviour for one stage then use this flag.

### 2.3 MQTT integration testing

#### 2.3.1 Testing with MQTT messages

Since version 0.4.0 Tavern has supported tests that require sending and receiving MQTT messages.

This is a very simple MQTT test that only uses MQTT messages:

```yaml
---

test_name: Test mqtt message response

paho-mqtt:
  client:
    transport: websockets
```

(continues on next page)
client_id: tavern-tester
connect
  host: localhost
  port: 9001
  timeout: 3

stages
  name: step 1 - ping/pong
  mqtt_publish
    topic: /device/123/ping
    payload: ping
  mqtt_response
    topic: /device/123/pong
    payload: pong
    timeout: 5

The first thing to notice is the extra paho-mqtt block required at the top level. When this block is present, an MQTT client will be started for the current test and is used to publish and receive messages from a broker.

**MQTT connection options**

The MQTT library used is the paho-mqtt Python library, and for the most part the arguments for each block are passed directly through to the similarly-named methods on the `paho.mqtt.client.Client` class.

The full list of options for the `mqtt` client block are listed below (host is the only required key, though you will almost always require some of the others):

- **client**: Passed through to `Client.__init__`.
  - transport: Connection type, optional. websockets or tcp. Defaults to tcp.
  - client_id: MQTT client ID, optional. Defaults to tavern-tester.
  - clean_session: Whether to connect with a clean session or not. true or false. Defaults to false.

- **connect**: Passed through to `Client.connect`.
  - host: MQTT broker host.
  - port: MQTT broker port. Defaults to 1883 in the paho-mqtt library.
  - keepalive: Keepalive frequency to MQTT broker. Defaults to 60 (seconds) in the paho-mqtt library. Note that some brokers will kick client off after 60 seconds by default (eg VerneMQ), so you might need to lower this if you are kicked off frequently.
  - timeout: How many seconds to try and connect to the MQTT broker before giving up. This is not passed through to paho-mqtt, it is implemented in Tavern. Defaults to 1.

- **tls**: Controls TLS connection - as well as enable, this accepts all keywords taken by `Client.tls_set()` (see paho documentation for the meaning of these keywords).
  - enable: Enable TLS connection with broker. If no other tls options are passed, using enable: true will enable tls without any custom certificates/keys/ciphers. If enable: false is used, any other tls options will be ignored.
  - ca_certs
  - certfile
  - keyfile
Tavern Documentation, Release 1.12.2

- cert_reqs
- tls_version
- ciphers
  - auth: Passed through to `Client.username_pw_set`.
    - username: Username to connect to broker with.
    - password: Password to use with username.

The above example connects to an MQTT broker on port 9001 using the websockets protocol, and will try to connect for 3 seconds before failing the test.

Similar to the persistent requests session, the MQTT client is created at the beginning of a test and used for all stages in the test.

### MQTT publishing options

Messages can be published using the MQTT broker with the `mqtt_publish` key. In the above example, a message is published on the topic `/device/123/ping`, with the payload `ping`.

Like when making HTTP requests, JSON can be sent using the `json` key instead of the `payload` key.

```yaml
mqtt_publish:
  topic: /device/123/ping
  json:
    thing_1: abc
    thing_2: 123
```

This will result in the MQTT payload `"thing_2": 123, "thing_1": "abc"` being sent.

The full list of keys for this block:

- `topic`: The MQTT topic to publish on
- `payload` OR `json`: A plain text payload to publish, or a YAML object to serialize into JSON.
- `qos`: QoS level for publishing. Defaults to 0 in paho-mqtt.

### Options for receiving MQTT messages

The `mqtt_response` key gives a topic and payload which should be received by the end of the test stage, or that stage will be considered a failure. This works by subscribing to the topic specified before running the test, and then waiting after the test for a specified timeout for that message to be sent. If a message on the topic specified with the same payload is not received within that timeout period, it is considered a failure.

If other messages on the same topic but with a different payload arrive in the meantime, they are ignored and a warning will be logged.

```yaml
mqtt_response:
  topic: /device/123/ping
  json:
    thing_1: abc
    thing_2: 123
```

The keys which can be used:

- `topic`: The MQTT topic to subscribe to
• **payload OR json**: A plain text payload or a YAML object that will be serialized into JSON that must match the payload of a message published to `topic`.

• **timeout**: How many seconds to wait for the message to arrive. Defaults to 3.

• **qos**: The level of QoS to subscribe to the topic with. This defaults to 1, and it is unlikely that you will need to ever set this value manually.

While the `json` key will follow the same matching rules as HTTP JSON responses, The special ‘anything’ token can be used with the `payload` key just to check that there was some response on a topic:

```yaml
mqtt_response:
  topic: /device/123/ping
  payload: !anything
```

Other type tokens such as `!anyint` will not work.

### 2.3.2 Mixing MQTT tests and HTTP tests

If the architecture of your program combines MQTT and HTTP, Tavern can seamlessly test either or both of them in the same test, and even in the same stage.

**MQTT messages in separate stages**

In this example we have a server that listens for an MQTT message from a device for it to say that a light has been turned on. When it receives this message, it updates a database so that each future request to get the state of the device will return the updated state.

```yaml
test_name: Make sure posting publishes mqtt message
includes:
  - !include common.yaml

# More realistic broker connection options
paho-mqtt: &mqtt_spec
  client:
    transport: websockets
    connect:
      host: an.mqtt.broker.com
      port: 4687
    tls:
      enable: true
    auth:
      username: joebloggs
      password: password123

stages:
  - name: step 1 - get device state with lights off
    request:
      url: "{host}/get_device_state"
      params:
        device_id: 123
      method: GET
      headers:
```

(continues on next page)
content-type: application/json
response
status_code: 200
json
lights: "off"
headers
content-type: application/json

- name: step 2 - publish an mqtt message saying that the lights are now on
mqtt_publish
topic: /device/123/lights
qos: 1
payload: "on"
delay_after: 2

- name: step 3 - get device state, lights now on
request
url: "\{(host)}/get_device_state"
params
device_id: 123
method: GET
headers
content-type: application/json
response
status_code: 200
json
lights: "on"
headers
content-type: application/json

You can see from this example that when using mqtt_publish we don’t necessarily need to expect a message to be published in return - We can just send a message and wait for it to be processed with delay_after.

**MQTT message in the same stage**

MQTT blocks and HTTP blocks can be combined in the same test stage to test that sending a HTTP request results in an MQTT message being sent.

Say we have a server that takes a device id and publishes an MQTT message to it saying hello:

```
---
test_name: Make sure posting publishes mqtt message
includes
- !include common.yaml
paho-mqtt: *mqtt_spec
stages
- name: step 1 - post message trigger
  request
  url: "\{(host)}/send_mqtt_message"
  json
  device_id: 123
  payload: "hello"
(continues on next page)```
method: POST
headers:
  content-type: application/json
response:
  status_code: 200
  json:
    topic: "/device/123"
  headers:
    content-type: application/json
mqtt_response:
  topic: /device/123
  payload: "hello"
  timeout: 5
  qos: 2

Before running the request in this stage, Tavern will subscribe to /device/123 with QoS level 2. After making the request (and getting the correct response from the server!), it will wait 5 seconds for a message to be published on that topic.

Note: You can only have one of request or mqtt_publish in a test stage. If you need to publish a message and send a HTTP request in sequence, use an approach like the previous example where they are in two separate stages.

2.4 Plugins

Since 0.10.0, Tavern has a simple plugin system which lets you change how requests are made. By default, all HTTP tests use the requests library and all MQTT tests use the paho-mqtt library.

However, there are some situations where you might not want to run tests against something other than a live server, or maybe you just want to use curl to extract some better usage statistics out of your requests. Tavern’s plugin system can be used to override this default behaviour (note however that it still ONLY supports HTTP and MQTT requests at the time of writing).

The best way to introduce the concepts for making a plugin is by using an example. For this we will be looking at a plugin used to run tests against a local flask server called tavern_flask.

2.4.1 The entry point

Plugins are loaded using two setuptools entry points, namely tavern_http for HTTP tests and tavern_mqtt for MQTT tests. The built-in requests and paho-mqtt functionality is implemented using plugins, so looking at the _plugins folder in the Tavern repository will also be useful as a reference when writing a plugin.

The entry point needs to point to either a class or a module which defines a preset number of variables.

Something like this should be in your setup.py or setup.cfg to make sure Tavern can pick it up at run time:

```python
# setup.cfg

# A http plugin. tavern_http is the entry point that Tavern searches for,
# 'requests' is the name of your plugin which is selected using the
# --tavern-http-backend command line flag. This points to a class in the
# tavernhook module.
tavern_http =
  requests = tavern._plugins.rest.tavernhook.TavernRestPlugin
```

(continues on next page)
Examples:

- The requests based http entry point points to a class using the module.submodule:member entry point syntax.
- The paho-mqtt plugin just uses a module using the module.submodule entry point syntax. This loads the schema from the file on import.
- The tavern-flask plugin also just uses a module.

### 2.4.2 Extra schema data

If your plugin needs extra metadata in each test to be able to make a request, extra schema data can be added with a schema key in your entry point. This should be a dictionary which is just merged into the base schema for tests.

There is currently only one key supported in the schema dictionary, initialisation. This defines a top level key in each test which your session or request classes can use to set up the test (see the mqtt documentation for an example of how this is used to connect to an MQTT broker).

Examples:

- The paho-mqtt plugin defines the client, connect, etc. keys which are used to connect to an MQTT broker.
- tavern-flask just requires a single key that points to the flask application that will be used to create a test client (see below).

### 2.4.3 Session type

session_type should return a class which describes a “session” which will be used throughout the entire test. It should be a class that fulfills two requirements:

1. It must take the same keyword arguments as the ‘base’ session object to create an instance for testing. For HTTP tests this is the same arguments as a requests.Session object, and for MQTT tests it is the same arguments as specified in the MQTT documentation. If your plugin does not support some of these arguments, raise a NotImplementedError which a short message explaining that it is not supported.
2. After creating the instance, it must be able to be used as a context manager. If you don’t need any functionality provided by this, you can define empty __enter__ and __exit__ methods on your class like so:

```python
class MySession(object):
    def __enter__(self):
        pass
    def __exit__(self, *args):
        pass
```

Examples:

- tavern-flask is fairly simple, it just creates a flask test client from the flask::app defined for the test (see schema documentation above) and dumps the body data for later use when making the request.
2.4.4 Request

request_type is a class that encapsulates the concept of a ‘request’ for your plugin. It takes 3 arguments:

- **session** is the session instance created as described above, *for that request type at that stage*. There may be multiple request types per test, but only one request is made per stage.
- **rspec** is a dictionary corresponding to the request at that stage. If you are writing a HTTP plugin, the dictionary will contain the keys as described in the http request documentation. If it is an MQTT plugin, it will contain keys described in the MQTT publish documentation.
- **test_block_config** is the global configuration for that test. At a minimum it will contain a key called variables, which contains all of the current variables that are available for formatting.

In the constructor, this request type should validate the input data and format the request variables given the test block config.

The class should also have a run method, which takes no arguments and is called to run the test. This should return some kind of class encapsulating response data which can be verified by your plugin’s response verifier class.

Tavern knows which request keyword (eg request, mqtt_publish) corresponds to your plugin by matching it to the plugin’s request_block_name. For the moment, this should be hardcoded to request for HTTP tests.

Examples:

- The base requests request object formats the keys and does some extra verification, such as logging a warning if a user tries to send a body with a GET request
- The paho-mqtt request formats the input data and just makes sure that a user is not trying to send two kinds of payloads at a time.
- tavern-flask reuses functionality from Tavern to format the keys and do extra verification.

2.4.5 Getting the expected response

get_expected_from_request should be a function that takes 3 arguments:

- **stage** is the entire test stage (ie, including the request block, test name, response block, etc) as a dictionary
- **test_block_config** is as above
- **session** is as above

This function should use this input data to calculate the expected response and perform any extra things that need doing based on the request or expected response. This will normally just be formatting the response block based on the variables in the test block config, but you may need to do extra things (such as subscribing to an MQTT topic).

Examples:

- The default behaviour is just to make sure that a correct response block is present and format the input data.
- An MQTT test requires that the client also checks to see if a response is expected and subscribes to the topic in question.
- tavern-flask behaves identically to the base Tavern behaviour.

2.4.6 Response

verifier_type is a class that encapsulates the concept of verifying a response for your plugin. It should inherit from tavern.response.base.BaseResponse, and take 4 arguments:

- **session** is as above
• **name** is the name of the test stage currently being run. This can be used for logging debug information.

• **expected** is the return value from `get_expected_from_request`.

• **test_block_config** is as above.

It should also define a couple of methods:

• **verify** takes one argument, which is the return value from the `run` method on your request class. It should read whatever information is relevant from this response object and verify that it is as expected, then return any values from the response which should be saved into the test block config. A plugin does not need to save anything - just return an empty dictionary if you don’t want to save anything. There are some utilities on `BaseResponse` to help with this, including printing errors and checking return values. This should raise a `tavern.exceptions.TestFailError` if verification fails. The easiest way to verify the response is to call `self._adderr` with a string to a list called `self.errors` for every error encountered. If there is anything in this dictionary at the end of `verify`, raise an exception.

• **__str__** should return a human-readable string describing the response. This is mainly for debugging, and should only give as much information as you think is required. For example, a HTTP response might be printed as (Like with a request, Tavern knows which verifier to use by looking at the `response_block_name` key.

Examples:

• The **base requests verifier** Checks a variety of things like the expected headers, expected redirect locations, cookies, etc.

• The **paho-mqtt** plugin needs to wait for the specified timeout to see if a message was received on a given topic. Note that there does not need to be a response for an MQTT request - a stage might consist of just an `mqtt_publish` block with no expected response.

• **tavern-flask** just reuses functionality from the base verifier again. Because the flask `Response` object is slightly different from the requests one, some conversion has to be done on the data.

### 2.5 Debugging a test

**This section assumes you’re using pytest to run tests.**

When making a test it’s not always going to work first time, and at the time of writing the error reporting is a bit messy because it shows the whole stack trace from pytest is printed out (which can be a few hundred lines, most of which is useless). Figuring out if it’s an error in the test, an error in the API response, or even a bug in Tavern can be a bit tricky.

#### 2.5.1 Setting up logging

Tavern has extensive debug logging to help figure out what is going on in tests. When running your tests, it helps a lot to set up logging so that you can check the logs in case something goes wrong. The easiest way to do this is with `dictConfig` from the Python logging library. It can also be useful to use `colorlog` to colourize the output so it’s easier to see the different log levels. An example logging configuration

```yaml
# log_spec.yaml

---

version: 1
formatters:
  default:
    # colorlog is really useful
```
Which is used like this:

```python
from logging import config
import yaml

with open("log_spec.yaml", "r") as log_spec_file:
    config.dictConfig(yaml.load(log_spec_file))
```

Making sure this code is called before running your tests (for example, by putting into `conftest.py`) will show the tavern logs if a test fails.

By default, recent versions of pytest will print out log messages in the “Captured stderr call” section of the output - if you have set up your own logging, you probably want to disable this by also passing `-p no:logging` to the invocation of pytest.

### 2.5.2 Setting pytest options

Some pytest options can be used to make the test output easier to read.

- Using the `-vv` option will show a separate line for each test and whether it has passed or failed as well as showing more information about mismatches in data returned vs data expected
- Using `--tb=short` will reduce the amount of data presented in the traceback when a test fails. If logging it set up as above, any important information will be present in the logs.
- If you just want to run one test you can use the `-k` flag to make pytest only run that test.

### 2.5.3 Example

Say we are running against the advanced example from Tavern but we have an error in the yaml:
# Log in ...

```yaml
- name: post a number
  request:
    url: "{host}/numbers"
    json:
      name: smallnumber
      number: 123
    method: POST
  headers:
    content-type: application/json
    Authorization: "bearer {test_login_token:s}"
  response:
    status_code: 201
    headers:
      content-type: application/json
    # This key will not actually be present in the response
    json:
      a_key: missing
```

Having full debug output can be a bit too much information, so we set up logging as above but at the INFO level rather than DEBUG.

We run this by doing `py.test --tb=short -p no:logging` and get the following output:

```bash
.....
...
..../....
init
  init
    __init__
      init = func()
.../
init
  lambda
    return CallInfo lambda __init__.py:617 in __
.../
init
  lambda
    return self.hookexec(self, self.nourlifiers + self.wrappers, kwags)
.../
init
  lambda
    hook __init__.py:222 in __
.../
init
  hook
    self.__init__.py:216 in __
.../
init
  lambda
  firstresult=hook.spec.opts.get('firstresult')
.../
init
  lambda
  return outcome.get_result()
.../
init
  hook __init__.py:76 in get_result()
  raise exc
```

(continues on next page)
When tavern tries to access `a_key` in the response it gets a `KeyError` (shown in the logs), and the TestFailError in the stack trace gives a more human-readable explanation as to why the test failed.

### 2.5.4 New traceback option

Though this does give a lot of information about exactly when and where a test failed, it’s not very easy to tell what input actually caused this error. Since 0.13.0, you can use the `tavern-beta-new-traceback` flag to give a much nicer output showing the original source code for the stage, the formatted stages that Tavern uses to send the request, and any format variables. This is used like any other Pytest flag:

1. Passed on on the command line using `--tavern-beta-new-traceback`
2. Put in the Pytest configuration file (pytest.ini, setup.cfg, …) `tavern-beta-new-traceback = True`

Rather than the Python traceback as shown above, we get an error output like this:

```
Format variables:
  tavern.env_vars.TEST_HOST = 'http://localhost:5003'
  first_part = 'nested'
  second_part = 'again'

Source test stage:
  - name: Make requests using environment variables
    request:
```

(continues on next page)
• Format variables shows all the variables which are used for formatting in that stage. Any variables which are missing will be highlighted in red.

• The source test stage is the raw source code for the stage from the input file. This is before anything has been done to it - no formatting, no anchors, no includes, etc.

• The formatted stage shows the stage at the point that Tavern will start to perform the request - all variables will be formatted (if present), all YAML anchors will be resolved, etc.

• The errors will show which exception caused this test to fail

This output style will become the default in version 1.0.

Note that this will only show when a test fails in a way that Tavern can handle, and it will not be shown on things like IOErrors on input files or unhandled errors.

If a test fails in a way that does not raise a TestFailError, it might be a bug in Tavern - if this happens, feel free to make an issue on the repo.

### 2.6 Examples

#### 2.6.1 1) The simplest possible test

To show you just how simple a Tavern test can be, here’s one which uses the JSON Placeholder API at jsonplaceholder.typicode.com. To try it, create a new file called `test_minimal.tavern.yaml` with the following:

```yaml
---
test_name: Get some fake data from the JSON placeholder API
stages:
  - name: Make sure we have the right ID
    request:
      url: https://jsonplaceholder.typicode.com/posts/1
```

(continues on next page)
Tavern Documentation, Release 1.12.2

(continued from previous page)

```
    method    GET
response:
    status_code: 200
    json:
      id: 1
```

Next, install Tavern if you have not already:

```
$ pip install tavern
```

In most circumstances you will be using Tavern with pytest but you can also run it using the Tavern command-line interface, tavern-ci, which is installed along with Tavern:

```
$ tavern-ci test_minimal.tavern.yaml
```

Run tavern-ci --help for more usage information.

Note that Tavern will only run tests from files whose names follow the pattern test_*.tavern.yaml (or test_*.tavern.yml) - for example, test_minimal.tavern.yaml, test_another.tavern.yml.

### 2.6.2 2) Testing a simple server

In this example we will create a server with a single route which doubles any number you pass it, and write some simple tests for it. You’ll see how simple the YAML-based syntax can be, and the three different ways you can run Tavern tests.

Here’s what such a server might look like:

```python
# server.py
from flask import Flask, jsonify, request
app = Flask(__name__)

@app.route("/double", methods=["POST"],)
def double_number():
    r = request.get_json()
    try:
        number = r["number"]
    except (KeyError, TypeError):
        return jsonify({"error": "no number passed"}), 400
    try:
        double = int(number) * 2
    except ValueError:
        return jsonify({"error": "a number was not passed"}), 400
    return jsonify({"double": double}), 200
```

Run the server using Flask:

```
$ export FLASK_APP=server.py
$ flask run
```

There are two key things to test here: first, that it successfully doubles numbers and second, that it returns the correct error codes and messages. To do this we will write two tests, one for the success case and one for the error case. Each
test can contain one or more stages, and each stage has a name, a request and an expected response.

```yaml
# test_server.tavern.yaml
...

test_name: Make sure server doubles number properly

stages:
  - name: Make sure number is returned correctly
    request:
      url: http://localhost:5000/double
      json:
        number: 5
        method: POST
      headers:
        content-type: application/json
    response:
      status_code: 200
      json:
        double: 10

...

test_name: Check invalid inputs are handled

stages:
  - name: Make sure invalid numbers don't cause an error
    request:
      url: http://localhost:5000/double
      json:
        number: dkfsd
        method: POST
      headers:
        content-type: application/json
    response:
      status_code: 400
      json:
        error: a number was not passed

  - name: Make sure it raises an error if a number isn't passed
    request:
      url: http://localhost:5000/double
      json:
        wrong_key: 5
        method: POST
      headers:
        content-type: application/json
    response:
      status_code: 400
      json:
        error: no number passed
```

The tests can be run in three different ways: from Python code, from the command line, or with pytest. The most common way is to use pytest. All three require Tavern to be installed.

If you run pytest in a folder containing `test_server.tavern.yaml` it will automatically find the file and run the tests. Otherwise, you will need to point it to the folder containing the integration tests or add it to `setup.cfg/tox.ini/etc` so that Pytest’s collection mechanism knows where to look.
The command line tool is useful for bash scripting, for example if you want to verify that an API is works before deploying it, or for cron jobs.

```bash
$ tavern-ci test_server.tavern.yaml
$ echo $?
0
```

The Python library allows you to include Tavern tests in deploy scripts written in Python, or for use with a continuous integration setup:

```python
from tavern.core import run

success = run("test_server.tavern.yaml").

if not success:
    print("Error running tests").
```

See the documentation section on global configuration for use of the second argument.

### 2.6.3 3) Multi-stage tests

The final example uses a more complex test server which requires the user to log in, save the token it returns and use it for all future requests. It also has a simple database so we can check that data we send to it is successfully returned.

Here is the example server we will be using.

To test this behaviour we can use multiple tests in a row, keeping track of variables between them, and ensuring the server state has been updated as expected.

```
test_name: Make sure server saves and returns a number correctly

stages:
    - name: login
      request:
        url: http://localhost:5000/login
        json:
          user: test-user
          password: correct-password
        method: POST
        headers:
          content-type: application/json
      response:
        status_code: 200
        json:
          $ext
```

(continues on next page)
function tavern.testutils.helpers:validate_jwt
extra_kwarg :
jwt_key "token"
key CGQgaG7GYvTcplaQ2qosLy4
options
verify_signature true
verify_aud false
headers
content-type application/json
save
json
  test_login_token token

- name post a number
request
  url http://localhost:5000/numbers
json
  name smallnumber
  number 123
method POST
headers
  content-type application/json
  Authorization "bearer {test_login_token:s}"
response
  status_code 201
json

headers:
  content-type application/json

- name Make sure its in the db
request
  url http://localhost:5000/numbers
params
  name smallnumber
method GET
headers
  content-type application/json
  Authorization "bearer {test_login_token:s}"
response
  status_code 200
json
  number 123
headers:
  content-type application/json

This example illustrates three major parts of the Tavern syntax: saving data, using that data in later requests and using validation functions.

### 2.6.4 Further reading

There are more examples in the examples folder on Github, showing how to do some more advanced testing, including how to test using MQTT. Tavern also has a lot of integration tests that show its behaviour - you might find it useful to check out the integration tests folder for some more examples.

To see the source code, suggest improvements or even contribute a pull request check out the GitHub repository.
2.7 Advanced Cookbook

This page contains some extra reading you might find useful when writing and running your Tavern tests.

2.7.1 Pytest plugins

Because Tavern is built upon Pytest, the majority of Pytest plugins can be used seamlessly to help your testing.

- pytest-sugar and pytest-tldr can all be used to make test result reporting more pretty or less pretty.
- pytest-instafail shows errors in line while tests are running
- pytest-html can be used to provide html reports of test runs
- pytest-xdist can be used to run your tests in parallel, speeding up test runs if you have a large number of tests

2.7.2 Using with docker

Tavern can be fairly easily used with Docker to run your integration tests. Simply use this Dockerfile as a base and add any extra requirements you need (such as any Pytest plugins as mentioned above):

```bash
# tavern.Dockerfile
FROM python:3.7-alpine
RUN pip3 install tavern
```

Build with:
```
```

Or if you need a specific version (hopefully you shouldn’t):

```bash
# tavern.Dockerfile
FROM python:3.7-alpine
ARG TAVERNVER
RUN pip3 install tavern==$TAVERNVER
```

```bash
export TAVERNVER=0.24.0
docker build --build-arg TAVERNVER=$TAVERNVER --file tavern.Dockerfile --tag tavern:__$TAVERNVER__ .
```

Note that if you do this in a folder with a lot of subfolders (for example, an npm project) you probably want to create a .dockerignore file so that the build does not take an incredibly long time to start up - see the documentation here for information on how to create one.

This can be used by running it on the command line with `docker run`, but it is often easier to use it in a docker-compose file like this:

```yaml
---
version: '3.4'
services
  tavern
```

(continues on next page)
build:
  context: .
image: tavern.Dockerfile
env_file
  # Any extra environment variables for testing
  # This will probably contain things like names of docker containers to run
  tests against
  required-env-keys.env
volumes
  # The folder that our integration tests are in
  ./integration_tests:/integration_tests
  # If you have anything in your pytest configuration it will also need mounting
  # here then pointing to with the -c flag to pytest
command
  python
  -m
  pytest
  # Point to any global configuration files
  -t tavern-global-cfg
  /integration_tests/local_urls.yaml
  # And any other flags you want to pass
  -p
  no:logging
  # And then point to the folder we mounted above
  /integration_tests

# Optionally also just run your application in a docker container as well
application:
build:
  context: .
image: application.Dockerfile
command:
  ...

2.7.3 Using marks with fixtures

Though passing arguments into fixtures is unsupported at the time of writing, you can use Pytest marks to control the behaviour of fixtures.

If you have a fixture that loads some information from a file or some other external data source, but the behaviour needs to change depending on which test is being run, this can be done by marking the test and accessing the test Node in your fixture to change the behaviour:

test_name: endpoint 1 test
marks:
  endpoint_1
  usefixtures:
    read_uuid
stages
...

test_name: endpoint 2 test

(continues on next page)
In the `read_uuid` fixture:

```python
import pytest
import json

@pytest.fixture
def read_uuid(request):  # 'request' is a built-in pytest fixture
    marks = request.node.own_markers
    mark_names = [m.name for m in marks]

    with open("stored_uuids.json", "r") as ufile:
        uuids = json.load(ufile)

    if "endpoint_1" in mark_names:
        return uuids["endpoint_1"]
    elif "endpoint_2" in mark_names:
        return uuids["endpoint_2"]
    else:
        pytest.fail("No marker found on test!")
```