pytest_httpserver Documentation

Release 0.3.7

Zsolt Cserna

Feb 14, 2021

Contents

pytest-httpserver is a python package which allows you to start a real HTTP server for your tests. The server can be configured programmatically to how to respond to requests.

This project aims to provide an easy to use API to start the server, configure the request handlers and then shut it down gracefully. All of these without touching a configuration file or dealing with daemons.

As the HTTP server is spawned in a different thread and listening on a TCP port, you can use any HTTP client. This library also helps you migrating to a different HTTP client library without the need to re-write any test for your client application.

This library can be used with pytest most conveniently but if you prefer to use other test frameworks, you can still use it with the context API or by writing a wrapper for it.

CHAPTER 1

Example

```
import requests

def test_json_client(httpserver: HTTPServer):
    httpserver.expect_request("/foobar").respond_with_json({"foo": "bar"})
    assert requests.get(httpserver.url_for("/foobar")).json() == {'foo': 'bar'}
```

For further details, please read the guide or the *API documentation*.

1.1 Tutorial

If you haven't worked with this library yet, this document is for you.

1.1.1 Writing your first test

With pytest-httpserver, a test looks like this:

```
import requests

def test_json_client(httpserver: HTTPServer):
    httpserver.expect_request("/foobar").respond_with_json({"foo": "bar"})
    assert requests.get(httpserver.url_for("/foobar")).json() == {'foo': 'bar'}
```

In the first line of the code, we are setting up an expectation. The expectation contains the http request which is expected to be made:

```
httpserver.expect_request("/foobar")
```

This code tells that the httpserver, which is started automatically and running on localhost, should accept the request "http://localhost/foobar". Configuring how to handle this request is then done with the following method:

```
respond_with_json({"foo": "bar"})
```

This tells that when the request arrives to the *http://localhost/foobar* URL, it must respond with the provided json. The library accepts here any python object which is json serializable. Here, a dict is provided.

In the next line, an http request is sent with the *requests* library:

```
assert requests.get(httpserver.url_for("/foobar")).json() == {'foo': 'bar'}
```

There's no customization (such as mocking) to be made. You don't need to figure out the port number where the server is running, as there's the `url_for()` method provided to format the URL.

As you can see there are two different part of the httpserver configuration:

1. setting up what kind of request we are expecting
2. telling how the request should be handled and which content should be responded.

Important note on server port number

The test should be run with an unprivileged user. As it is not possible to bind to the default http port (80), the library binds the server to an available port which is higher than 1024. In the examples on this page when we are referring to the url *http://localhost/...* it is assumed that the url contains the http port also.

It is advised to use the `url_for()` method to construct an URL as it will always contain the correct port number in the URL.

If you need the http port as an integer, you can get is by the `port` attribute of the `httpserver` object.

1.1.2 How to test your http client

Note: This section describes the various ways of http client testing. If you are sure that pytest-httpserver is the right library for you, you can skip this section.

You've written your first http client application and you want to write a test for it. You have the following options:

1. Test your application against the production http server
2. Mock your http calls, so they won't reach any real server
3. Run a fake http server listening on localhost behaving like the real http server

pytest-httpserver provides API for the 3rd option: it runs a real http server on localhost so you can test your client connecting to it.

However, there's no silver bullet and the possibilities above have their pros and cons.

Test your application against the production http server

Pros:

- It needs almost no change in the source code and you can run the tests with no issues.
- Writing tests is simple.

Cons:

- The tests will use a real connection to the real server, it will generate some load on the server, which may be acceptable or not. If the real server is down or you have some connectivity issue, you can't run tests.
- If the server has some state, for example, a backend database with user data, authentication, etc, you have to solve the *shared resource* problem if you want to allow multiple test runnings on different hosts. For example, if there are more than one developers and/or testers.
- Ensuring that there's no crosstalk is very important: if there's some change made by one instance, it should be invisible to the other. It should either revert the changes or do it in a separate namespace which will be cleaned up by some other means such as periodic jobs. Also, the test should not have inconsistent state behind.

Mock your http calls, so they won't reach any real server

Pros:

- It needs almost no change in the source code and you can run the tests with no issues.
- There are excellent libraries supporting mocking such as **responses** and **pytest-vcr**.
- No need to ensure crosstalk or manage shared resources.
- Tests work offline.

Cons:

- No actual http requests are sent. It needs great effort to mock the existing behavior of the original library (such as **requests**) and you need to keep the two libraries in sync.
- Mocking must support the http client library of your choice. Eg. if you use **requests** you need to use **responses**. If you are using different libraries, the complexity raises.
- At some point, it is not like black-box testing as you need to know the implementation details of the original code.
- It is required to set up the expected requests and their responses. If the server doesn't work like your setup, the code will break when it is run with the real server.

Run a fake http server listening on localhost

Pros:

- Writing tests is simple.
- No need to ensure crosstalk or manage shared resources.
- Tests work offline.
- Actual http requests are sent. There's a real http server running speaking http protocol so you can test all the special cases you need. You can customize every http request expectations and their responses to the end.
- Testing connectivity issues is possible.
- There's no mocking, no code injection or class replacement.
- It is black-box testing as there's no need to know anything about the original code.

Cons:

- Some code changes required in the original source code. The code should accept the server endpoint (host and port) as a parameter or by some means of configuration. This endpoint will be set to localhost during the test running. If it is not possible, you need to tweak name resolution.

- It is required to set up the expected requests and their responses. If the server doesn't work like your setup, the code will break when it is run with the real server.
- Setting up TLS/SSL requires additional knowledge (cert generation, for example)

1.1.3 Specifying the expectations and constraints

In the above code, the most simple case was shown. The library provides many ways to customize the expectations.

In the example above, the code expected a request to */foobar* with any method (such as *GET*, *PUT*, *POST*, *DELETE*). If you want to limit the method to the *GET* method only, you can specify:

```
httpserver.expect_request("/foobar", method="GET")
```

Similarly, specifying the query parameters is possible:

```
httpserver.expect_request("/foobar", query_string="user=user1", method="GET")
```

This will match the GET request made to the http://localhost/foobar?user=user1 URL. If more constraint is specified to the `expect_request()` method, the expectation will be narrower, eg. it is similar when using logical AND.

If you want, you can specify the query string as a dictionary so the order of the key-value pairs does not matter:

```
httpserver.expect_request("/foobar", query_string={"user": "user1", "group": "group1"}
↪, method="GET")
```

Similar to query parameters, it is possible to specify constraints for http headers also.

For many parameters, you can specify either string or some expression (such as the dict in the example above).

For example, specifying a regexp pattern for the URI Is also possible by specifying a compiled regexp object:

```
httpserver.expect_request(re.compile("^/foo"), query_string={"user": "user1", "group
↪": "group1"}, method="GET")
```

The above will match every URI starting with "/foo".

All of these are documented in the *API documentation*.

1.1.4 Specifying responses

Once you have set up the expected request, it is required to set up the response which will be returned to the client.

In the example we used `respond_with_json()` but it is also possible to respond with an arbitrary content.

```
respond_with_data("Hello world!", content_type="text/plain")
```

In the example above, we are responding a text/plain content. You can specify the status also:

```
respond_with_data("Not found", status=404, content_type="text/plain")
```

With this method, it is possible to set the response headers, mime type.

In some cases you need to create your own Response instance (which is the Response object from the underlying werkzeug library), so you can respond with it. This allows more customization, however, in most cases the respond_with_data is sufficient:

```
respond_with_response(Response("Hello world!"))
# same as
respond_with_data("Hello world!"))
```

If you need to produce dynamic content, use the `respond_with_handler` method, which accepts a callable (eg. a python function):

```
def my_handler(request):
    # here, examine the request object
    return Response("Hello world!")

respond_with_handler(my_handler)
```

1.1.5 Ordered and oneshot expectations

In the above examples, we used `expect_request()` method, which registered the request to be handled. During the test running you can issue requests to this endpoint as many times as you want, and you will get the same response (unless you used the `respond_with_handler()` method, detailed above).

There are two other additional limitations which can be used:

- ordered handling, which specifies the order of the requests
- oneshot handling, which specifies the lifetime of the handlers for only one request

Ordered handling

The ordered handling specifies the order of the requests. It must be the sam as the order of the registration:

```
def test_ordered(httpserver: HTTPServer):
    httpserver.expect_ordered_request("/foobar").respond_with_data("OK foobar")
    httpserver.expect_ordered_request("/foobaz").respond_with_data("OK foobaz")

    requests.get(httpserver.url_for("/foobar"))
    requests.get(httpserver.url_for("/foobaz"))
```

The above code passes the test running. The first request matches the first handler, and the second request matches the second one.

When making the requests in a reverse order, it will fail:

```
def test_ordered(httpserver: HTTPServer):
    httpserver.expect_ordered_request("/foobar").respond_with_data("OK foobar")
    httpserver.expect_ordered_request("/foobaz").respond_with_data("OK foobaz")

    requests.get(httpserver.url_for("/foobaz"))
    requests.get(httpserver.url_for("/foobar")) # <- fail?
```

If you run the above code you will notice that no test failed. This is because the http server is running in its own thread, separately from the client code. It has no way to raise an assertion error in the client thread.

However, this test checks nothing but runs two subsequent queries and that's it. Checking the http status code would make it fail:

```
def test_ordered(httpserver: HTTPServer):
    httpserver.expect_ordered_request("/foobar").respond_with_data("OK foobar")
    httpserver.expect_ordered_request("/foobaz").respond_with_data("OK foobaz")

    assert requests.get(httpserver.url_for("/foobaz")).status_code == 200
    assert requests.get(httpserver.url_for("/foobar")).status_code == 200 # <- fail!
```

For further details about error handling, please read the *Handling test errors* chapter.

Oneshot handling

Oneshot handling is useful when you want to ensure that the client makes only one request to the specified URI. Once the request is handled and the response is sent, the handler is no longer registered and a further call to the same URL will be erroneous.

```
def test_oneshot(httpserver: HTTPServer):
    httpserver.expect_oneshot_request("/foobar").respond_with_data("OK")

    requests.get(httpserver.url_for("/foobar"))
    requests.get(httpserver.url_for("/foobar")) # this will get http status 500
```

If you run the above code you will notice that no test failed. This is because the http server is running in its own thread, separately from the client code. It has no way to raise an assertion error in the client thread.

However, this test checks nothing but runs two subsequent queries and that's it. Checking the http status code would make it fail:

```
def test_oneshot(httpserver: HTTPServer):
    httpserver.expect_oneshot_request("/foobar").respond_with_data("OK")

    assert requests.get(httpserver.url_for("/foobar")).status_code == 200
    assert requests.get(httpserver.url_for("/foobar")).status_code == 200 # fail!
```

For further details about error handling, please read the *Handling test errors* chapter.

Handling test errors

If you look at carefully at the test running, you realize that the second request (and all further requests) will get an http status 500 code, explaining the issue in the response body. When a properly written http client gets http status 500, it should raise an exception, which will be unhandled and in the end the test will be failed.

In some cases, however, you want to make sure that everything is ok so far, and raise AssertionError when something is not good. Call the `check_assertions()` method of the httpserver object, and this will look at the server's internal state (which is running in the other thread) and if there's something not right (such as the order of the requests not matching, or there was a non-matching request), it will raise an AssertionError and your test will properly fail:

```
def test_ordered_ok(httpserver: HTTPServer):
    httpserver.expect_ordered_request("/foobar").respond_with_data("OK foobar")
    httpserver.expect_ordered_request("/foobaz").respond_with_data("OK foobaz")

    requests.get(httpserver.url_for("/foobaz"))
    requests.get(httpserver.url_for("/foobar")) # gets 500

    httpserver.check_assertions() # this will raise AssertionError and make the test␣
↪failing
```

The server writes a log about the requests and responses which were processed. This can be accessed in the *log* attribute of the http server. This log is a python list with 2-element tuples (request, response).

Server lifetime

Http server is started when the first test uses the *httpserver* fixture, and it will be running for the rest of the session. The server is not stopped and started between the tests as it is an expensive operation, it takes up to 1 second to properly stop the server.

To avoid crosstalk (eg one test leaving its state behind), the server's state is cleaned up between test runnings.

Debugging

If you having multiple requests for the server, adding the call to `check_assertions()` may to debug as it will make the test failed as soon as possible.

```
import requests

def test_json_client(httpserver: HTTPServer):
    httpserver.expect_request("/foobar").respond_with_json({"foo": "bar"})
    requests.get(httpserver.url_for("/foo"))
    requests.get(httpserver.url_for("/bar"))
    requests.get(httpserver.url_for("/foobar"))

    httpserver.check_assertions()
```

In the above code, the first request (to **/foo**) is not successful (it gets http status 500), but as the response status is not checked (or any of the response), and there's no call to `check_assertions()`, the test continues the running. It gets through the **/bar** request, which is also not successful (and gets http status 500 also like the first one), then goes the last request which is successful (as there's a handler defined for it)

In the end, when checking the check_assertions() raise the error for the first request, but it is a bit late: figuring out the request which caused the problem could be troublesome. Also, it will report the problem for the first request only.

Adding more call of `check_assertions()` will help.

```
import requests

def test_json_client(httpserver: HTTPServer):
    httpserver.expect_request("/foobar").respond_with_json({"foo": "bar"})
    requests.get(httpserver.url_for("/foo"))
    httpserver.check_assertions()

    requests.get(httpserver.url_for("/bar"))
    httpserver.check_assertions()

    requests.get(httpserver.url_for("/foobar"))
    httpserver.check_assertions()
```

In the above code, the test will fail after the first request.

In case you do not want to fail the test, you can use any of these options:

- `assertions` attribute of the `httpserver` object is a list of the known errors. If it is non-empty, then there was an issue.

- `format_matchers()` method of the `httpserver` object returns which handlers have been registered to the server. In some cases, registering non-matching handlers causes the problem so printing this string can help to diagnose the problem.

1.1.6 Advanced topics

This is the end of the tutorial, however, not everything is covered here and this library offers a lot more.

Further readings:

- *API documentation*
- *Howto*

1.2 Howto

This documentation is a collection of the most common use cases, and their solutions. If you have not used this library before, it may be better to read the *Tutorial* first.

1.2.1 Matching query parameters

To match query parameters, you must not included them to the URI, as this will not work:

```
def test_query_params(httpserver):
    httpserver.expect_request("/foo?user=bar") # never do this
```

There's an explicit place where the query string should go:

```
def test_query_params(httpserver):
    httpserver.expect_request("/foo", query_string="user=bar")
```

The `query_string` is the parameter which does not contains the leading question mark `?`.

Note: The reason behind this is the underlying http server library *werkzeug*, which provides the `Request` object which is used for the matching the request with the handlers. This object has the `query_string` attribute which contains the query.

As the order of the parameters in the query string usually does not matter, you can specify a dict for the `query_string` parameter (the naming may look a bit strange but we wanted to keep API compatibility and this dict matching feature was added later).

```
def test_query_params(httpserver):
    httpserver.expect_request("/foo", query_string={"user": "user1", "group": "group1
↪"}).respond_with_data("OK")

    assert requests.get("/foo?user=user1&group=group1").status_code == 200
    assert requests.get("/foo?group=group1&user=user1").status_code == 200
```

In the example above, both requests pass the test as we specified the expected query string as a dictionary.

Behind the scenes an additional step is done by the library: it parses up the query_string into the dict and then compares it with the dict provided.

1.2.2 URI matching

The simplest form of URI matching is providing as a string. This is a equality match, if the URI of the request is not equal with the specified one, the request will not be handled.

If this is not desired, you can specify a regexp object (returned by the `re.compile()` call).

```
httpserver.expect_request(re.compile("^/foo"), method="GET")
```

The above will match every URI starting with "/foo".

There's an additional way to extend this functionality. You can specify your own method which will receive the URI. All you need is to subclass from the `URIPattern` class and define the `match()` method which will get the uri as string and should return a boolean value.

```
class PrefixMatch(URIPattern):
    def __init__(self, prefix: str):
        self.prefix = prefix

    def match(self, uri):
        return uri.startswith(self.prefix)

def test_uripattern_object(httpserver: HTTPServer):
    httpserver.expect_request(PrefixMatch("/foo")).respond_with_json({"foo": "bar"})
```

1.2.3 Authentication

When doing http digest authentication, the client may send a request like this:

```
GET /dir/index.html HTTP/1.0
Host: localhost
Authorization: Digest username="Mufasa",
                     realm="testrealm@host.com",
                     nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                     uri="/dir/index.html",
                     qop=auth,
                     nc=00000001,
                     cnonce="0a4f113b",
                     response="6629fae49393a05397450978507c4ef1",
                     opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Implementing a matcher is difficult for this request as the order of the parameters in the `Authorization` header value is arbitrary.

By default, pytest-httpserver includes an Authorization header parser so the order of the parameters in the `Authorization` header does not matter.

```
def test_authorization_headers(httpserver: HTTPServer):
    headers_with_values_in_direct_order = {
        'Authorization': ('Digest username="Mufasa",'
                          'realm="testrealm@host.com",'
                          'nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",'
                          'uri="/dir/index.html",'
                          'qop=auth,'
                          'nc=00000001,'
                          'cnonce="0a4f113b",'
                          'response="6629fae49393a05397450978507c4ef1",'
```

(continues on next page)

(continued from previous page)

```
                        'opaque="5ccc069c403ebaf9f0171e9517f40e41"')
    }
    httpserver.expect_request(uri='/', headers=headers_with_values_in_direct_order).
↪respond_with_data('OK')
    response = requests.get(httpserver.url_for('/'), headers=headers_with_values_in_
↪direct_order)
    assert response.status_code == 200
    assert response.text == 'OK'

    headers_with_values_in_modified_order = {
        'Authorization': ('Digest qop=auth,'
                          'username="Mufasa",'
                          'nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",'
                          'uri="/dir/index.html",'
                          'nc=00000001,'
                          'realm="testrealm@host.com",'
                          'response="6629fae49393a05397450978507c4ef1",'
                          'cnonce="0a4f113b",'
                          'opaque="5ccc069c403ebaf9f0171e9517f40e41"')
    }
    response = requests.get(httpserver.url_for('/'), headers=headers_with_values_in_
↪modified_order)
    assert response.status_code == 200
    assert response.text == 'OK'
```

1.2.4 JSON matching

Matching the request data can be done in two different ways. One way is to provide a python string (or bytes object) whose value will be compared to the request body.

When the request contains a json, matching to will be error prone as an object can be represented as json in different ways, for example when different length of indentation is used.

To match the body as json, you need to add the python data structure (which could be dict, list or anything which can be the result of *json.loads()* call). The request's body will be loaded as json and the result will be compared to the provided object. If the request's body cannot be loaded as json, the matcher will fail and *pytest-httpserver* will proceed with the next registered matcher.

Example:

```
def test_json_matcher(httpserver: HTTPServer):
    httpserver.expect_request("/foo", json={"foo": "bar"}).respond_with_data("Hello␣
↪world!")
    resp = requests.get(httpserver.url_for("/foo"), json={"foo": "bar"})
    assert resp.status_code == 200
    assert resp.text == "Hello world!"
```

Note: JSON requests usually come with `Content-Type: application/json` header. *pytest-httpserver* provides the *headers* parameter to match the headers of the request, however matching json body does not imply matching the *Content-Type* header. If matching the header is intended, specify the expected *Content-Type* header and its value to the headers parameter.

Note: *json* and *data* parameters are mutually exclusive so both of then cannot be specified as in such case the behavior

is ambiguous.

Note: The request body is decoded by using the *data_encoding* parameter, which is default to *utf-8*. If the request comes in a different encoding, and the decoding fails, the request won't match with the expected json.

1.2.5 Advanced header matching

For each http header, you can specify a callable object (eg. a python function) which will be called with the header name, header actual value and the expected value, and will be able to determine the matching.

You need to implement such a function and then use it:

```
def case_insensitive_matcher(header_name: str, actual: str, expected: str) -> bool:
    if header_name == "X-Foo":
        return actual.lower() == expected.lower()
    else:
        return actual == expected


def test_case_insensitive_matching(httpserver: HTTPServer):
    httpserver.expect_request("/", header_value_matcher=case_insensitive_matcher,␣
↪headers={"X-Foo": "bar"}).respond_with_data("OK")

    assert requests.get(httpserver.url_for("/"), headers={"X-Foo": "bar"}).status_
↪code == 200
    assert requests.get(httpserver.url_for("/"), headers={"X-Foo": "BAR"}).status_
↪code == 200
```

Note: Header value matcher is the basis of the `Authorization` header parsing.

If you want to change the matching of only one header, you may want to use the `HeaderValueMatcher` class.

In case you want to do it globally, you can add the header name and the callable to the `HeaderValueMatcher.DEFAULT_MATCHERS` dict.

```
from pytest_httpserver import HeaderValueMatcher

def case_insensitive_compare(actual: str, expected: str) -> bool:
    return actual.lower() == expected.lower()

HeaderValueMatcher.DEFAULT_MATCHERS["X-Foo"] = case_insensitive_compare

def test_case_insensitive_matching(httpserver: HTTPServer):
    httpserver.expect_request("/", headers={"X-Foo": "bar"}).respond_with_data("OK")

    assert requests.get(httpserver.url_for("/"), headers={"X-Foo": "bar"}).status_
↪code == 200
    assert requests.get(httpserver.url_for("/"), headers={"X-Foo": "BAR"}).status_
↪code == 200
```

In case you don't want to change the defaults, you can provide the `HeaderValueMatcher` object itself.

```
from pytest_httpserver import HeaderValueMatcher

def case_insensitive_compare(actual: str, expected: str) -> bool:
    return actual.lower() == expected.lower()

def test_own_matcher_object(httpserver: HTTPServer):
    matcher = HeaderValueMatcher({"X-Bar": case_insensitive_compare})

    httpserver.expect_request("/", headers={"X-Bar": "bar"}, header_value_
↪matcher=matcher).respond_with_data("OK")

    assert requests.get(httpserver.url_for("/"), headers={"X-Bar": "bar"}).status_
↪code == 200
    assert requests.get(httpserver.url_for("/"), headers={"X-Bar": "BAR"}).status_
↪code == 200
```

1.2.6 Customizing host and port

By default, the server run by pytest-httpserver will listen on localhost on a random available port. In most cases it works well as you want to test your app in the local environment.

If you need to change this behavior, there are a plenty of options. It is very important to make these changes before starting the server, eg. before running any test using the httpserver fixture.

Use IP address *0.0.0.0* to listen globally.

> **Warning:** You should be careful when listening on a non-local ip (such as *0.0.0.0*). In this case anyone knowing your machine's IP address and the port can connect to the server.

Environment variables

Set `PYTEST_HTTPSERVER_HOST` and/or `PYTEST_HTTPSERVER_PORT` environment variables to the desired values.

Class attributes

Changing `HTTPServer.DEFAULT_LISTEN_HOST` and `HTTPServer.DEFAULT_LISTEN_PORT` attributes. Make sure that you do this before running any test requiring the `httpserver` fixture. One ideal place for this is putting it into `conftest.py`.

Fixture

Overriding the `httpserver_listen_address` fixture. Similar to the solutions above, this needs to be done before starting the server (eg. before referencing the `httpserver` fixture).

```
import pytest

@pytest.fixture
def httpserver_listen_address():
    return ("127.0.0.1", 8000)
```

1.2.7 Multi-threading support

When your client runs in a thread, everything completes without waiting for the first response. To overcome this problem, you can wait until all the handlers have been served or there's some error happened.

This is available only for oneshot and ordered handlers, as permanent handlers last forever.

To have this feature enabled, use the context object returned by the `wait()` method of the `httpserver` object.

This method accepts the following parameters:

- raise_assertions: whether raise assertions on unexpected request or timeout or not
- stop_on_nohandler: whether stop on unexpected request or not
- timeout: time (in seconds) until time is out

Behind the scenes it synchronizes the state of the server with the main thread.

Last, you need to assert on the `result` attribute of the context object.

```
def test_wait_success(httpserver: HTTPServer):
    waiting_timeout = 0.1

    with httpserver.wait(stop_on_nohandler=False, timeout=waiting_timeout) as waiting:
        requests.get(httpserver.url_for("/foobar"))
        httpserver.expect_oneshot_request("/foobar").respond_with_data("OK foobar")
        requests.get(httpserver.url_for("/foobar"))
    assert waiting.result

    httpserver.expect_oneshot_request("/foobar").respond_with_data("OK foobar")
    httpserver.expect_oneshot_request("/foobaz").respond_with_data("OK foobaz")
    with httpserver.wait(timeout=waiting_timeout) as waiting:
        requests.get(httpserver.url_for("/foobar"))
        requests.get(httpserver.url_for("/foobaz"))
    assert waiting.result
```

In the above code, all the request.get() calls could be in a different thread, eg. running in parallel, but the exit condition of the context object is to wait for the specified conditions.

1.2.8 Emulating connection refused error

If by any chance, you want to emulate network errors such as *Connection reset by peer* or *Connection refused*, you can simply do it by connecting to a random port number where no service is listening:

```
import pytest
import requests

def test_connection_refused():
    # assumes that there's no server listening at localhost:1234
    with pytest.raises(requests.exceptions.ConnectionError):
        requests.get("http://localhost:1234")
```

However connecting to the port where the httpserver had been started will still succeed as the server is running continuously. This is working by design as starting/stopping the server is costly.

```
import pytest
import requests

```

(continues on next page)

(continued from previous page)

```
# setting a fixed port for httpserver
@pytest.fixture
def httpserver_listen_address():
    return ("127.0.0.1", 8000)

# this test will pass
def test_normal_connection(httpserver):
    httpserver.expect_request("/foo").respond_with_data("foo")
    assert requests.get("http://localhost:8000/foo").text == "foo"


# this tess will FAIL, as httpserver started in test_normal_connection is
# still running
def test_connection_refused():
    with pytest.raises(requests.exceptions.ConnectionError):
        # this won't get Connection refused error as the server is still
        # running.
        # it will get HTTP status 500 as the handlers registered in
        # test_normal_connection have been removed
        requests.get("http://localhost:8000/foo")
```

To solve the issue, the httpserver can be stopped explicitly. It will start implicitly when the first test starts to use it. So the `test_connection_refused` test can be re-written to this:

```
def test_connection_refused(httpserver):
    httpserver.stop() # stop the server explicitly
    with pytest.raises(requests.exceptions.ConnectionError):
        requests.get("http://localhost:8000/foo")
```

1.2.9 Emulating timeout

To emulate timeout, there's one way to register a handler function which will sleep for a given amount of time.

```
import time
from pytest_httpserver import HTTPServer
import pytest
import requests


def sleeping(request):
    time.sleep(2)  # this should be greater than the client's timeout parameter


def test_timeout(httpserver: HTTPServer):
    httpserver.expect_request("/baz").respond_with_handler(sleeping)
    with pytest.raises(requests.exceptions.ReadTimeout):
        assert requests.get(httpserver.url_for("/baz"), timeout=1)
```

There's one drawback though: the test takes 2 seconds to run as it waits the handler thread to be completed.

1.3 API documentation

1.3.1 pytest_httpserver

This is package provides the main API for the pytest_httpserver package.

HTTPServer

class pytest_httpserver.**HTTPServer**(*host='localhost', port=0, ssl_context: Optional[ssl.SSLContext] = None, default_waiting_settings: Optional[pytest_httpserver.httpserver.WaitingSettings] = None*)

Server instance which manages handlers to serve pre-defined requests.

Parameters

- **host** – the host or IP where the server will listen
- **port** – the TCP port where the server will listen
- **ssl_context** – the ssl context object to use for https connections
- **default_waiting_settings** – the waiting settings object to use as default settings for *wait()* context manager

log

Attribute containing the list of two-element tuples. Each tuple contains `Request` and `Response` object which represents the incoming request and the outgoing response which happened during the lifetime of the server.

no_handler_status_code

Attribute containing the http status code (int) which will be the response status when no matcher is found for the request. By default, it is set to *500* but it can be overridden to any valid http status code such as *404* if needed.

add_assertion(*obj*)

Add a new assertion

Assertions can be added here, and when *check_assertions()* is called, it will raise AssertionError for pytest with the object specified here.

Parameters **obj** – An object which will be passed to AssertionError.

application(*request: werkzeug.wrappers.request.Request*)

Entry point of werkzeug.

This method is called for each request, and it then calls the undecorated *dispatch()* method to serve the request.

Parameters **request** – the request object from the werkzeug library

Returns the response object what the dispatch returned

check_assertions()

Raise AssertionError when at least one assertion added

The first assertion added by *add_assertion()* will be raised and it will be removed from the list.

This method can be useful to get some insights into the errors happened in the sever, and to have a proper error reporting in pytest.

clear()
Clears and resets the state attributes of the object.

This method is useful when the object needs to be re-used but stopping the server is not feasible.

clear_all_handlers()
Clears all types of the handlers (ordered, oneshot, permanent)

clear_assertions()
Clears the list of assertions

clear_log()
Clears the list of log entries

create_matcher(**args*, ***kwargs*) → pytest_httpserver.httpserver.RequestMatcher
Creates a `RequestMatcher` instance with the specified parameters.

This method can be overridden if you want to use your own matcher.

dispatch(*request: werkzeug.wrappers.request.Request*) → werkzeug.wrappers.response.Response
Dispatch a request to the appropriate request handler.

This method tries to find the request handler whose matcher matches the request, and then calls it in order to serve the request.

First, the request is checked for the ordered matchers. If there's an ordered matcher, it must match the request, otherwise the server will be put into a *permanent failure* mode in which it makes all request failed - this is the intended way of working of ordered matchers.

Then oneshot handlers, and the permanent handlers are looked up.

Parameters **request** – the request object from the werkzeug library

Returns the response object what the handler responded, or a response which contains the error

expect_oneshot_request(*uri: Union[str, pytest_httpserver.httpserver.URIPattern, Pattern[str]], method: str = '__ALL', data: Union[str, bytes, None] = None, data_encoding: str = 'utf-8', headers: Optional[Mapping[str, str]] = None, query_string: Union[None, pytest_httpserver.httpserver.QueryMatcher, str, bytes, Mapping[KT, VT_co]] = None, header_value_matcher: Optional[pytest_httpserver.httpserver.HeaderValueMatcher] = None, json: Any = <UNDEFINED>*) → pytest_httpserver.httpserver.RequestHandler
Create and register a oneshot request handler.

This is a method for convenience. See *expect_request()* for documentation.

Parameters

- **uri** – URI of the request. This must be an absolute path starting with `/`, a `URIPattern` object, or a regular expression compiled by `re.compile()`.
- **method** – HTTP method of the request. If not specified (or *METHOD_ALL* specified), all HTTP requests will match.
- **data** – payload of the HTTP request. This could be a string (utf-8 encoded by default, see *data_encoding*) or a bytes object.
- **data_encoding** – the encoding used for data parameter if data is a string.
- **headers** – dictionary of the headers of the request to be matched
- **query_string** – the http query string, after `?`, such as `username=user`. If string is specified it will be encoded to bytes with the encode method of the string. If dict is specified, it will be matched to the `key=value` pairs specified in the

request. If multiple values specified for a given key, the first value will be used. If multiple values needed to be handled, use `MultiDict` object from werkzeug.
- **header_value_matcher** – *HeaderValueMatcher* that matches values of headers.
- **json** – a python object (eg. a dict) whose value will be compared to the request body after it is loaded as json. If load fails, this matcher will be failed also. *Content-Type* is not checked. If that's desired, add it to the headers parameter.

Returns Created and register *RequestHandler*.

Parameters *json* and *data* are mutually exclusive.

expect_ordered_request(*uri: Union[str, pytest_httpserver.httpserver.URIPattern, Pattern[str]], method: str = '__ALL', data: Union[str, bytes, None] = None, data_encoding: str = 'utf-8', headers: Optional[Mapping[str, str]] = None, query_string: Union[None, pytest_httpserver.httpserver.QueryMatcher, str, bytes, Mapping[KT, VT_co]] = None, header_value_matcher: Optional[pytest_httpserver.httpserver.HeaderValueMatcher] = None, json: Any = <UNDEFINED>*) → pytest_httpserver.httpserver.RequestHandler

Create and register a ordered request handler.

This is a method for convenience. See *expect_request()* for documentation.

Parameters
- **uri** – URI of the request. This must be an absolute path starting with `/`, a `URIPattern` object, or a regular expression compiled by `re.compile()`.
- **method** – HTTP method of the request. If not specified (or *METHOD_ALL* specified), all HTTP requests will match.
- **data** – payload of the HTTP request. This could be a string (utf-8 encoded by default, see *data_encoding*) or a bytes object.
- **data_encoding** – the encoding used for data parameter if data is a string.
- **headers** – dictionary of the headers of the request to be matched
- **query_string** – the http query string, after `?`, such as `username=user`. If string is specified it will be encoded to bytes with the encode method of the string. If dict is specified, it will be matched to the `key=value` pairs specified in the request. If multiple values specified for a given key, the first value will be used. If multiple values needed to be handled, use `MultiDict` object from werkzeug.
- **header_value_matcher** – *HeaderValueMatcher* that matches values of headers.
- **json** – a python object (eg. a dict) whose value will be compared to the request body after it is loaded as json. If load fails, this matcher will be failed also. *Content-Type* is not checked. If that's desired, add it to the headers parameter.

Returns Created and register *RequestHandler*.

Parameters *json* and *data* are mutually exclusive.

expect_request(*uri: Union[str, pytest_httpserver.httpserver.URIPattern, Pattern[str]], method: str = '__ALL', data: Union[str, bytes, None] = None, data_encoding: str = 'utf-8', headers: Optional[Mapping[str, str]] = None, query_string: Union[None, pytest_httpserver.httpserver.QueryMatcher, str, bytes, Mapping[KT, VT_co]] = None, header_value_matcher: Optional[pytest_httpserver.httpserver.HeaderValueMatcher] = None, handler_type: pytest_httpserver.httpserver.HandlerType = <HandlerType.PERMANENT: 'permanent'>, json: Any = <UNDEFINED>*) → pytest_httpserver.httpserver.RequestHandler

Create and register a request handler.

If *handler_type* is *HandlerType.PERMANENT* a permanent request handler is created. This handler can be used as many times as the request matches it, but ordered handlers have higher priority so if there's one or more ordered handler registered, those must be used first.

If *handler_type* is *HandlerType.ONESHOT* a oneshot request handler is created. This handler can be only used once. Once the server serves a response for this handler, the handler will be dropped.

If *handler_type* is *HandlerType.ORDERED* an ordered request handler is created. Comparing to oneshot handler, ordered handler also determines the order of the requests to be served. For example if there are two ordered handlers registered, the first request must hit the first handler, and the second request must hit the second one, and not vice versa. If one or more ordered handler defined, those must be exhausted first.

Parameters

- **uri** – URI of the request. This must be an absolute path starting with `/`, a `URIPattern` object, or a regular expression compiled by `re.compile()`.
- **method** – HTTP method of the request. If not specified (or *METHOD_ALL* specified), all HTTP requests will match. Case insensitive.
- **data** – payload of the HTTP request. This could be a string (utf-8 encoded by default, see *data_encoding*) or a bytes object.
- **data_encoding** – the encoding used for data parameter if data is a string.
- **headers** – dictionary of the headers of the request to be matched
- **query_string** – the http query string, after `?`, such as `username=user`. If string is specified it will be encoded to bytes with the encode method of the string. If dict is specified, it will be matched to the `key=value` pairs specified in the request. If multiple values specified for a given key, the first value will be used. If multiple values needed to be handled, use `MultiDict` object from werkzeug.
- **header_value_matcher** – *HeaderValueMatcher* that matches values of headers.
- **handler_type** – type of handler
- **json** – a python object (eg. a dict) whose value will be compared to the request body after it is loaded as json. If load fails, this matcher will be failed also. *Content-Type* is not checked. If that's desired, add it to the headers parameter.

Returns Created and register *RequestHandler*.

Parameters *json* and *data* are mutually exclusive.

format_matchers() → str

Return a string representation of the matchers

This method returns a human-readable string representation of the matchers registered. You can observe which requests will be served, etc.

This method is primarily used when reporting errors.

is_running() → bool

Returns *True* when the server is running, otherwise *False*.

respond_nohandler(*request: werkzeug.wrappers.request.Request*)

Add a 'no handler' assertion.

This method is called when the server wasn't able to find any handler to serve the request. As the result, there's an assertion added (which can be raised by *check_assertions()*).

respond_permanent_failure()

Add a 'permanent failure' assertion.

This assertion means that no further requests will be handled. This is the resuld of missing an ordered matcher.

start()
Start the server in a thread.

This method returns immediately (e.g. does not block), and it's the caller's responsibility to stop the server (by calling *stop()*) when it is no longer needed).

If the sever is not stopped by the caller and execution reaches the end, the program needs to be terminated by Ctrl+C or by signal as it will not terminate until the thread is stopped.

If the sever is already running `HTTPServerError` will be raised. If you are unsure, call *is_running()* first.

There's a context interface of this class which stops the server when the context block ends.

stop()
Stop the running server.

Notifies the server thread about the intention of the stopping, and the thread will terminate itself. This needs about 0.5 seconds in worst case.

Only a running server can be stopped. If the sever is not running, :py:class'HTTPServerError' will be raised.

thread_target()
This method serves as a thread target when the server is started.

This should not be called directly, but can be overridden to tailor it to your needs.

url_for(*suffix: str*)
Return an url for a given suffix.

This basically means that it prepends the string `http://$HOST:$PORT/` to the *suffix* parameter (where $HOST and $PORT are the parameters given to the constructor).

Parameters **suffix** – the suffix which will be added to the base url. It can start with / (slash) or not, the url will be the same.

Returns the full url which refers to the server

wait(*raise_assertions: Optional[bool] = None*, *stop_on_nohandler: Optional[bool] = None*, *timeout: Optional[float] = None*)
Context manager to wait until the first of following event occurs: all ordered and oneshot handlers were executed, unexpected request was received (if *stop_on_nohandler* is set to *True*), or time was out

Parameters

- **raise_assertions** – whether raise assertions on unexpected request or timeout or not
- **stop_on_nohandler** – whether stop on unexpected request or not
- **timeout** – time (in seconds) until time is out

Example:

```
def test_wait(httpserver):
    httpserver.expect_oneshot_request('/').respond_with_data('OK')
    with httpserver.wait(raise_assertions=False, stop_on_
↪nohandler=False, timeout=1) as waiting:
        requests.get(httpserver.url_for('/'))
    # `waiting` is :py:class:`Waiting`
    assert waiting.result
    print('Elapsed time: {} sec'.format(waiting.elapsed_time))
```

WaitingSettings

class pytest_httpserver.**WaitingSettings**(*raise_assertions: bool = True*, *stop_on_nohandler: bool = True*, *timeout: float = 5*)

Class for providing default settings and storing them in HTTPServer

Parameters

- **raise_assertions** – whether raise assertions on unexpected request or timeout or not
- **stop_on_nohandler** – whether stop on unexpected request or not
- **timeout** – time (in seconds) until time is out

HeaderValueMatcher

class pytest_httpserver.**HeaderValueMatcher**(*matchers: Optional[Mapping[str, Callable[[str, str], bool]]] = None*)

Matcher object for the header value of incoming request.

Parameters **matchers** – mapping from header name to comparator function that accepts actual and expected header values and return whether they are equal as bool.

RequestHandler

class pytest_httpserver.**RequestHandler**(*matcher: pytest_httpserver.httpserver.RequestMatcher*)

Represents a response function and a `RequestHandler` object.

This class connects the matcher object with the function responsible for the response.

Parameters **matcher** – the matcher object

respond(*request: werkzeug.wrappers.request.Request*) → werkzeug.wrappers.response.Response

Calls the request handler registered for this object.

If no request handler was specified previously, it raises `NoHandlerError` exception.

Parameters **request** – the incoming request object

Returns the response object

respond_with_data(*response_data: Union[str, bytes] = ''*, *status: int = 200*, *headers: Optional[Mapping[str, str]] = None*, *mimetype: Optional[str] = None*, *content_type: Optional[str] = None*)

Registers a respond handler function which responds raw data.

For detailed description please see the `Response` object as the parameters are analogue.

Parameters

- **response_data** – a string or bytes object representing the body of the response
- **status** – the HTTP status of the response
- **headers** – the HTTP headers to be sent (excluding the Content-Type header)
- **content_type** – the content type header to be sent
- **mimetype** – the mime type of the request

respond_with_handler(*func: Callable[[werkzeug.wrappers.request.Request], werkzeug.wrappers.response.Response]*)

Registers the specified function as a responder.

The function will receive the request object and must return with the response object.

respond_with_json(*response_json*, *status: int = 200*, *headers: Optional[Mapping[str, str]] = None*, *content_type: str = 'application/json'*)

Registers a respond handler function which responds with a serialized JSON object.

Parameters

- **response_json** – a JSON-serializable python object
- **status** – the HTTP status of the response
- **headers** – the HTTP headers to be sent (excluding the Content-Type header)
- **content_type** – the content type header to be sent

respond_with_response(*response: werkzeug.wrappers.response.Response*)

Registers a respond handler function which responds the specified response object.

Parameters **response** – the response object which will be responded

1.3.2 pytest_httpserver.httpserver

This module contains some internal classes which are normally not instantiated by the user.

class pytest_httpserver.httpserver.**RequestMatcher**(*uri: Union[str, pytest_httpserver.httpserver.URIPattern, Pattern[str]], method: str = '__ALL', data: Union[str, bytes, None] = None, data_encoding: str = 'utf-8', headers: Optional[Mapping[str, str]] = None, query_string: Union[None, pytest_httpserver.httpserver.QueryMatcher, str, bytes, Mapping[KT, VT_co]] = None, header_value_matcher: Optional[pytest_httpserver.httpserver.HeaderValueMatcher] = None, json: Any = <UNDEFINED>*)

Matcher object for the incoming request.

It defines various parameters to match the incoming request.

Parameters

- **uri** – URI of the request. This must be an absolute path starting with `/`, a `URIPattern` object, or a regular expression compiled by `re.compile()`.
- **method** – HTTP method of the request. If not specified (or *METHOD_ALL* specified), all HTTP requests will match.
- **data** – payload of the HTTP request. This could be a string (utf-8 encoded by default, see *data_encoding*) or a bytes object.
- **data_encoding** – the encoding used for data parameter if data is a string.
- **headers** – dictionary of the headers of the request to be matched
- **query_string** – the http query string, after `?`, such as `username=user`. If string is specified it will be encoded to bytes with the encode method of the string. If dict is specified, it will be matched to the `key=value` pairs specified in the request. If multiple values specified for a given key, the first value will be used. If multiple values needed to be handled, use `MultiDict` object from werkzeug.

difference(*request: werkzeug.wrappers.request.Request*) → list
Calculates the difference between the matcher and the request.

Returns a list of fields where there's a difference between the request and the matcher. The returned list may have zero or more elements, each element is a three-element tuple containing the field name, the request value, and the matcher value.

If zero-length list is returned, this means that there's no difference, so the request matches the fields set in the matcher object.

match(*request: werkzeug.wrappers.request.Request*) → bool
Returns whether the request matches the parameters set in the matcher object or not. *True* value is returned when it matches, *False* otherwise.

match_data(*request: werkzeug.wrappers.request.Request*) → bool
Matches the data part of the request

Parameters **request** – the HTTP request

Returns *True* when the data is matched or no matching is required. *False* otherwise.

match_json(*request: werkzeug.wrappers.request.Request*) → bool
Matches the request data as json.

Load the request data as json and compare it to self.json which is a json-serializable data structure (eg. a dict or list).

Parameters **request** – the HTTP request

Returns *True* when the data is matched or no matching is required. *False* otherwise.

class pytest_httpserver.httpserver.**Error**
Base class for all exception defined in this package.

class pytest_httpserver.httpserver.**NoHandlerError**
Raised when a `RequestHandler` has no registered method to serve the request.

class pytest_httpserver.httpserver.**HTTPServerError**
Raised when there's a problem with HTTP server.

class pytest_httpserver.httpserver.**RequestHandlerList**
Represents a list of `RequestHandler` objects.

match(*request: werkzeug.wrappers.request.Request*) → pytest_httpserver.httpserver.RequestHandler
Returns the first request handler which matches the specified request. Otherwise, it returns *None*.

1.4 Background

This document describes what design decisions were made during the development of this library. It also describes how the library works in detail.

This document assumes that you can use the library and have at least limited knowledge about the source code. If you feel that it is not true for you, you may want to read the *Tutorial* and *Howto*.

1.4.1 API design

The API should be simple for use to simple cases, but also provide great flexibility for the advanced cases. When increasing flexibility of the API it should not change the simple API unless it is absolutely required.

API compatibility is paramount. API breaking is only allowed when it is on par with the the the gain of the new functionality.

Adding new parameters to functions which have default value is not considered a breaking API change.

Simple API

API should be kept as simple as possible. It means that describing an expected request and its response should be trivial for the user. For this reason, the API is flat: it contains a handful of functions which have many parameters accepting built-in python types (such as bytes, string, int, etc) in contrast to more classes and functions with less arguments.

This API allows to define an expected request and the response which will be sent back to the client in a single line. This is one of the key features so using the library is not complicated.

Example:

```
def test_query_params(httpserver):
    httpserver.expect_request("/foo", query_string={"user": "user1"}).respond_with_
↪data("OK")
```

It is simple in the most simple cases, but once the expectation is more specific, the line can grow significantly, so here the user is expected to put the literals into variables:

```
def test_query_params(httpserver):
    httpserver.expect_request("/foo", query_string=expected_query).respond_with_data(
↪"OK")
```

If the user wants something more complex, classes are available for this which can be instantiated and then specified for the parameters normally accepting only built-in types.

The easy case should be made easy, with the possibility of making advanced things in a bit more complex way.

Flexible API

The API should be also made flexible as possible but it should not break the simple API and not make the simple API complicated. A good example for this is the *respond_with_handler* method, which accepts a callable object (eg. a function) which receives the request object and returns the response object.

The user can implement the required logic there.

Adding this flexibility however did not cause any change in the simple API, the simple cases can be still used as before.

Higher-level API

In the early days of this library, it wanted to support the low-level http protocol elements: request status, headers, etc to provide full coverage for the protocol itself. This was made in order to made the most advanced customizations possible.

Then the project received a few PRs adding *HeaderValueMatcher* and support for authorization which relied on the low-level API to add a higher-level API without breaking it. In the opposite case, adding a low-level API to a high-level would not be possible.

Transparency

The API provided by *pytest-httpserver* is transparent. That means that the objects (most importantly the *Request* and *Response* objects) defined by *werkzeug* are visible by the user of *pytest-httpserver*, there is no wrapping made. This is done by the sake of simplicity.

As *werkzeug* provides a stable API, there's no need to change this in the future, however this also limits the library to stick with *werkzeug* in the long term. Replacing *werkzeug* to something else would break the API due to this transparency.

1.4.2 Requirements

This section describes how to work with pytest-httpserver's requirements. These are the packages used by the library.

Number of requirements

It is required to keep the requirements at minimum. When adding a new library to the package requirements, research in the following topics should be done:

- code quality
- activity of the development and maintenance
- number of open issues, and their content
- how many people using that library
- python interpreter versions supported
- amount of API breaking changes
- license

Sometimes, it is better to have the own implementation instead of having a tiny library added to the requirements, which may cause compatibility issues.

Requirements version restrictions

In general, the package requirements should have no version restrictions. For example, the *werkzeug* library has no restrictions, which means that if a new version comes out of it, it is assumed that *pytest-httpserver* will be able to run with it.

Many people uses this library in an environment having full of other packages and limiting version here will limit their versions in their requirements also. For example if there's a software using *werkzeug 1.0.0* and our requirements have *<0.9* specified it will make *pytest-httpserver* incompatible with their software.

Requirements testing

Currently it is required to test with only the latest version of the required packages. However, if there's an API breaking change which affects *pytest-httpserver*, a decision should be made:

- apply version restrictions, possibly making *pytest-httpserver* incompatible with some other software
- add workaround to *pytest-httpserver* to support both APIs

1.4.3 HTTP server

The chosen HTTP server which drives this library is imlemented by the *werkzeug* library. The reason behind this decision is that *werkzeug* is used by Flask, a very popular web framework and it provides a proven, stable API in the long term.

1.4.4 Supported python versions

Supporting the latest python versions (such as 3.7 and 3.8 at the time of writing this), is a must. Supporting the older versions is preferred, following the state of the officially supported python versions by PSF.

The library should be tested periodically on the supported versions.

Dropping support for old python versions is possible if supporting would cause an issue or require extensive workaround. Currently, 3.4 is still supported by the library, however it is deprecated by PSF. As it causes no problems for *pytest-httpserver* (there's an additional requirement for this in the setup.py, but that's all), the support for this version will be maintained as long as possible. Once a new change is added to the library which require great effort to maintain compatibility with 3.4, the support for it will be dropped.

1.4.5 Testing and coverage

It is not required to have 100% test coverage but all possible use-cases should be covered. Github actions is used to test the library on all the supported python versions, and tox.ini is provided if local testing is desired.

When a bug is reported, there should be a test for it, which would re-produce the error and it should pass with the fix.

1.4.6 Server starting and stopping

The server is started when the first test is run which uses the httpserver fixture. It will be running till the end of the session, and new tests will use the same instance. A cleanup is done between the tests which restores the clean state (no handlers registered, empty log, etc) to avoid cross-talk.

The reason behind this is the time required to stop the server. For some reason, *werkzeug* (the http server used) needs about 1 second to stop itself. Adding this time to each test is not acceptable in most of the cases.

Note that it is still compatible with *pytest-xdist* (a popular pytest extension to run the tests in parallel) as in such case, distinct test sessions will be run and those will have their own http server instance.

1.4.7 Fixture scope

Due to the nature of the http server (it is run only once), it seems to be a good recommendation to keep the httpserver fixture session scoped, not function scoped. The problem is that the cleanup which needs to be done between the tests (as the server is run only once, see above), and that cleanup needs to be attached to a function scoped fixture.

1.4.8 HTTP port selection

In early versions of the library, the user had to specify which port the server should be bound. This later changed to have an so-called ephemeral port, which is a random free port number chosen by the kernel. It is good because it guarantees that it will be available and it allows parallel test runnings for example.

In some cases it is not desired (eg if the code being tested has wired-in port number), in such cases it is still possible to specify the port number.

Also, the host can be specified which allows to bind on "0.0.0.0" so the server is accessible from the network in case you want to test a javascript code running on a different server in a browser.

1.5 Release Notes

1.5.1 0.3.7

Other Notes

- Removed pytest-runner from setup.py as it is deprecated and makes packaging inconvenient as it needs to be installed before running setup.py.

1.5.2 0.3.6

New Features

- HTTP methods are case insensitive. The HTTP method specified is converted to uppercase in the library.
- It is now possible to specify a JSON-serializable python value (such as dict, list, etc) and match the request to it as JSON. The request's body is loaded as JSON and it will be compared to the expected value.
- The http response code sent when no handler is found for the request can be changed. It is set to 500 by default.

1.5.3 0.3.5

New Features

- Extend URI matching by allowing to specify URIPattern object or a compiled regular expression, which will be matched against the URI. URIPattern class is defined as abstract in the library so the user need to implement a new class based on it.

1.5.4 0.3.4

Bug Fixes

- Fix the tests assets created for SSL/TLS tests by extending their expiration time. Also update the Makefile which can be used to update these assets.

1.5.5 0.3.3

New Features

- Besides bytes and string, dict and MultiDict objects can be specified as query_string. When these objects are used, the query string gets parsed into a dict (or MultiDict), and comparison is made accordingly. This enables the developer to ignore the order of the keys in the query_string when expecting a request.

Bug Fixes

- Fixed issue #16 by converting string object passed as query_string to bytes which is the type of the query string in werkzeug, and also allowing bytes as the parameter.
- Fix release tagging. 0.3.2 was released in a mistake by tagging 3.0.2 to the branch.

Other Notes

- Add more files to source distribution (sdist). It now contains tests, assets, examples and other files.

1.5.6 0.3.1

New Features

- Add httpserver_listen_address fixture which is used to set up the bind address and port of the server. Setting bind address and port is possible by overriding this fixture.

1.5.7 0.3.0

New Features

- Support ephemeral port. This can be used by specify 0 as the port number to the HTTPServer instance. In such case, an unused port will be picked up and the server will start listening on that port. Querying the port attribute after server start reveals the real port where the server is actually listening.
- Unify request functions of the HTTPServer class to make the API more straightforward to use.

Upgrade Notes

- The default port has been changed to 0, which results that the server will be staring on an ephemeral port.
- **The following methods of HTTPServer have been changed in a backward-incompatible way:**
  - *pytest_httpserver.HTTPServer.expect_request()* becomes a general function accepting handler_type parameter so it can create any kind of request handlers
  - *pytest_httpserver.HTTPServer.expect_oneshot_request()* no longer accepts the ordered parameter, and it creates an unordered oneshot request handler
  - *pytest_httpserver.HTTPServer.expect_ordered_request()* is a new method craeting an ordered request handler

1.5.8 0.2.2

New Features

- Make it possible to intelligently compare headers. To accomplish that HeaderValueMatcher was added. It already contains logic to compare unknown headers and authorization headers. Patch by Roman Inflianskas.

1.5.9 0.2.1

Prelude

Minor fixes in setup.py and build environment. No actual code change in library .py files.

1.5.10 0.2

New Features

- When using pytest plugin, specifying the bind address and bind port can also be possible via environment variables. Setting PYTEST_HTTPSERVER_HOST and PYTEST_HTTPSERVER_PORT will change the bind host and bind port, respectively.
- SSL/TLS support added with using the SSL/TLS support provided by werkzeug. This is based on the ssl module from the standard library.

1.5.11 0.1.1

Prelude

Minor fixes in setup.py and build environment. No actual code change in library .py files.

1.5.12 0.1

Prelude

First release

1.6 Upgrade guides

The following document describes how to upgrade to a given version of the library which introduces breaking changes.

1.6.1 Introducing breaking changes

When a breaking change is about to be made in the library, an intermediate release is released which generates deprecation warnings when the functionality to be removed is used. This does not break any functionality but shows a warning instead.

Together with this intermediate release, a new *pre-release* is released to *pypi*. This release removes the functionality described by the warning, but *pip* does not install this version unless you specify the *–pre* parameter to *pip install*.

Once you made the required changes to make your code compatible with the new version, you can install the new version by *pip install –pre pytest-httpserver*.

After a given time period, a new non-pre release is released, this will be installed by pip similar to other releases and it will break your code if you have not made the required changes. If this happens, you can still pin the version in requirements.txt or other places. Usually specifying the version with == operator fixes the version, but for more details please read the documentation of the tool you are using in manage dependencies.

1.6.2 1.0.0

In pytest-httpserver 1.0.0 the following breaking changes were made.

- The scope of `httpserver_listen_address` fixture changed from **function** to **session**

In order to make your code compatible with the new version of pytest-httpserver, you need to specify the *session* scope explicitly.

Example

Old code:

```
import pytest

@pytest.fixture
def httpserver_listen_address():
    return ("127.0.0.1", 8888)
```

New code:

```
import pytest

@pytest.fixture(scope="session")
def httpserver_listen_address():
    return ("127.0.0.1", 8888)
```

As this fixture is now defined with session scope, it will be called only once, when it is first referenced by a test or by another fixture.

Note: There were other, non-breaking changes introduced to 1.0.0. For details, please read the *Release Notes*.

Python Module Index

p

Index

R

S

T

U

W