pytest_httpserver Documentation

Release 0.2.1

Zsolt Cserna

Oct 28, 2018

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.

The aim of this project is 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 in the most convenient way 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 *User's Guide* or the *API documentation*.

1.1 User's Guide

1.1.1 Starting and stopping

The server can be started by instatiating it and then calling the *pytest_httpserver.HTTPServer.start()* method. This will start the server in a separate thread, so you will need to make sure that the *pytest_httpserver.HTTPServer.stop()* method is called before your code exits.

A free TCP port needs to be specified when instantiating the server, where no other daemon is listening.

If you are using the pytest plugin it is done automatically by the plugin. Possibility to change the TCP port is TBD.

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.

If pytest plugin is not used, the DEFAULT_LISTEN_HOST and DEFAULT_LISTEN_PORT class attributes can be set on the HTTPServer class.

1.1.2 Configuring

By configuring the server means registering handlers for specific requests. Once a request matches with the configuration the specified response handler is fired and the reponse is served.

Requests

When registering a `pytest_httpserver.server.RequestMatcher`, it can use various parts of the HTTP request to be matched: URI, method, data, headers, and query string can be specified. All of these are based on simple qeuality checking, with the exception of method and URI where a special value specifying *any* can be given (variables *URI_DEFAULT* and *METHOD_ALL*, respectively).

`pytest_httpserver.server.HTTPServer` also determines how these matchers are looked up and what their lifetime is. You can register handlers which handle any amount of requests, but you can also register one-shot handlers which only handle one request and then they disappear.

Also, there's ordered handlers which also specify the order of the requests to be handled. Not matching the order of their registration, the server will refuse to serve any further requests.

With all of these, you can create a server with very permissive to very strict request handling.

Responses

Once the request is matched with one of the matchers, the handler gets fired, which can return a static response or you can create a function which can return a dynamic response. When dealing with static responses you can determine all parts of the http response (status, headers, content, etc), and you can also specify a JSON-serializable object to be returned as a json.

1.1.3 Debugging errors while testing

When the tests are running against the server and no matcher can be found for the given request, the server replies with HTTP status 500, and a short error text. This is not very helpful in most cases so if you want to check what is the situation, you should call *pytest_httpserver.HTTPServer.format_matchers()* or *pytest_httpserver.HTTPServer.check_assertions()* methods. The first one returns a human-readable string representation of the matchers registered. The second one raises *AssertionError* with the errors happened during the testing in the server.

Also there's a *pytest_httpserver.HTTPServer.log* attribute which contains the request-reponse object pairs what the server handled.

1.2 API documentation

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

class pytest_httpserver.**HTTPServer**(*host='localhost'*, *port=4000*, *ssl_context: Optional[ssl.SSLContext] = 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

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.

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*)
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*) → werkzeug.wrappers.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: str*, *method: str = '__ALL'*, *data: Union[str, bytes, None] = None*, *data_encoding: str = 'utf-8'*, *headers: Optional[Mapping[str, str]] = None*, *query_string: Optional[str] = None*, ***, *ordered=False*) → pytest_httpserver.httpserver.RequestHandler
Create and register a oneshot request handler.

This handler can be only used once. Once the server serves a response for this handler, the handler will be dropped.

Ordered handler (when *ordered* parameter is *True*) 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 vica 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 `/`.
- **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 starting with `?`, such as `?username=user`
- **ordered** – specifies whether to create an ordered handler or not. See above for details.

Returns Created and register `RequestHandler`.

expect_request(*uri: str*, *method: str = '__ALL'*, *data: Union[str, bytes, None] = None*, *data_encoding: str = 'utf-8'*, *headers: Optional[Mapping[str, str]] = None*, *query_string: Optional[str] = None*) → pytest_httpserver.httpserver.RequestHandler

Create and register a permanent request handler.

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.

Parameters

- **uri** – URI of the request. This must be an absolute path starting with `/`.
- **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
- **ordered** – specifies whether to create an ordered handler or not. See above for details.

Returns Created and register `RequestHandler`.

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 primairly used when reporting errors.

is_running() → bool

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

respond_nohandler(*request: werkzeug.wrappers.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 thred is stopped.

If the sever is already running :py:class'HTTPServerError' will be raised. If you are unsure, call :py:meth'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 runnig, :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 overriden 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

class pytest_httpserver.httpserver.**RequestMatcher**(*uri: str*, *method: str = '__ALL'*, *data: Union[str, bytes, None] = None*, *data_encoding: str = 'utf-8'*, *headers: Optional[Mapping[str, str]] = None*, *query_string: Optional[str] = None*)

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 /.

- **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 starting with `?`, such as `?username=user`

difference(*request: werkzeug.wrappers.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*) → 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*) → 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.

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.**RequestMatcher**(*uri: str*, *method: str = '__ALL'*, *data: Union[str, bytes, None] = None*, *data_encoding: str = 'utf-8'*, *headers: Optional[Mapping[str, str]] = None*, *query_string: Optional[str] = None*)
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 `/`.
- **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 starting with `?`, such as `?username=user`

difference(*request: werkzeug.wrappers.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*) → 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*) → 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.

class pytest_httpserver.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*) → werkzeug.wrappers.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], werkzeug.wrappers.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*)
Registers a respond handler function which responds the specified response object.

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

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

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

1.3 Release Notes

1.3.1 0.2.1

Prelude

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

1.3.2 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.

1.3.3 0.1.1

Prelude

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

1.3.4 0.1

Prelude

First release

Python Module Index

p

Index

S

T

U