pytest-flask Documentation

Release 1.0.1.dev44+g04c1255

Vital Kudzelka

Feb 18, 2023

Contents

Pytest-flask is a plugin for pytest that provides a set of useful tools to test Flask applications and extensions.

CHAPTER 1

Quickstart

Install plugin via `pip`:

```
pip install pytest-flask
```

Define your application fixture in `conftest.py`:

```
from myapp import create_app

@pytest.fixture
def app():
    app = create_app()
    return app
```

Now you can use the `app` fixture in your test suite. You can run your tests with:

```
pytest
```

CHAPTER 2

User's Guide

This part of the documentation will show you how to get started in using pytest-flask with your application.

2.1 Getting started

Pytest is capable to pick up and run existing tests without any or little configuration. This section describes how to get started quickly.

2.1.1 Step 1. Install

`pytest-flask` is available on PyPi, and can be easily installed via `pip`:

```
pip install pytest-flask
```

2.1.2 Step 2. Configure

Define your application fixture in `conftest.py`:

```
from myapp import create_app

@pytest.fixture
def app():
    app = create_app()
    return app
```

2.1.3 Step 3. Run your test suite

Use the `pytest` command to run your test suite:

```
pytest
```

Note: Test discovery.

Pytest discovers your tests and has a built-in integration with other testing tools (such as `nose`, `unittest` and `doctest`). More comprehensive examples and use cases can be found in the official documentation.

2.1.4 What's next?

The *Feature reference* section gives a more detailed view of available features, as well as test fixtures and markers.

Consult the pytest documentation for more information about pytest itself.

If you want to contribute to the project, see the *How to contribute* section.

2.2 Feature reference

Extension provides some sugar for your tests, such as:

- Access to context bound objects (`url_for`, `request`, `session`) without context managers:

```
def test_app(client):
    assert client.get(url_for('myview')).status_code == 200
```

- Easy access to `JSON` data in response:

```
@api.route('/ping')
def ping():
    return jsonify(ping='pong')

def test_api_ping(client):
    res = client.get(url_for('api.ping'))
    assert res.json == {'ping': 'pong'}
```

Note: User-defined `json` attribute/method in application response class will not be overwritten. So you can define your own response deserialization method:

```
from flask import Response
from myapp import create_app

class MyResponse(Response):
    '''Implements custom deserialization method for response objects.'''
    @property
    def json(self):
        return 42

@pytest.fixture(scope="session")
def app():
    app = create_app()
    app.response_class = MyResponse
    return app
```

(continues on next page)

(continued from previous page)

```
def test_my_json_response(client):
    res = client.get(url_for('api.ping'))
    assert res.json == 42
```

- Running tests in parallel with pytest-xdist. This can lead to significant speed improvements on multi core/multi CPU machines.

  This requires the `pytest-xdist` plugin to be available, it can usually be installed with:

```
pip install pytest-xdist
```

  You can then run the tests by running:

```
pytest -n <number of processes>
```

Not enough pros? See the full list of available fixtures and markers below.

2.2.1 Fixtures

`pytest-flask` provides a list of useful fixtures to simplify application testing. More information on fixtures and their usage is available in the pytest documentation.

2.2.1.1 `client` - application test client

An instance of `app.test_client`. Typically refers to flask.Flask.test_client.

Hint: During test execution a request context will be automatically pushed for you, so context-bound methods can be conveniently called (e.g. `url_for`, `session`.

Example:

```
def test_myview(client):
    assert client.get(url_for('myview')).status_code == 200
```

2.2.1.2 `client_class` - application test client for class-based tests

Example:

```
@pytest.mark.usefixtures('client_class')
class TestSuite:

    def test_myview(self):
        assert self.client.get(url_for('myview')).status_code == 200
```

2.2.1.3 `config` - application config

An instance of `app.config`. Typically refers to flask.Config.

2.2.1.4 `live_server` - application live server

Run application in a separate process (useful for tests with Selenium and other headless browsers).

Hint: The server's URL can be retrieved using the `url_for` function.

```
from flask import url_for

@pytest.mark.usefixtures('live_server')
class TestLiveServer:

    def test_server_is_up_and_running(self):
        res = urllib2.urlopen(url_for('index', _external=True))
        assert b'OK' in res.read()
        assert res.code == 200
```

`--start-live-server` - start live server automatically (default)

`--no-start-live-server` - don't start live server automatically

By default the server will start automatically whenever you reference `live_server` fixture in your tests. But starting live server imposes some high costs on tests that need it when they may not be ready yet. To prevent that behaviour pass `--no-start-live-server` into your default options (for example, in your project's `pytest.ini` file):

```
[pytest]
addopts = --no-start-live-server
```

Note: You **should manually start** live server after you finish your application configuration and define all required routes:

```
def test_add_endpoint_to_live_server(live_server):
    @live_server.app.route('/test-endpoint')
    def test_endpoint():
        return 'got it', 200

    live_server.start()

    res = urlopen(url_for('test_endpoint', _external=True))
    assert res.code == 200
    assert b'got it' in res.read()
```

`--live-server-wait` - the live server wait timeout (5 seconds)

The timeout after which test case is aborted if live server is not started.

`--live-server-port` - use a fixed port

By default the server uses a random port. In some cases it is desirable to run the server with a fixed port. You can use `--live-server-port` (for example, in your project's `pytest.ini` file):

```
[pytest]
addopts = --live-server-port=5000
```

2.2.1.5 `request_ctx` - request context (Deprecated)

This fixture is deprecated and will be removed in the future.

The request context which contains all request relevant information.

Hint: The request context has been pushed implicitly any time the `app` fixture is applied and is kept around during test execution, so it's easy to introspect the data:

```
from flask import request, url_for

def test_request_headers(client):
    res = client.get(url_for('ping'), headers=[('X-Something', '42')])
    assert request.headers['X-Something'] == '42'
```

`live_server_scope` - set the scope of the live server

By default, the server will be scoped to `session` for performance reasons, however if your server has global state and you want better test isolation, you can use the `live_server_scope` ini option to change the fixture scope:

```
[pytest]
live_server_scope = function
```

2.2.1.6 HTTP Request

Common request methods are available through the internals of the Flask API. Specifically, the API creates the default flask.Flask.test_client instance, which works like a regular Werkzeug test client.

Examples:

```
def test_post_request(client, live_server):
    @live_server.app.route('/load-data')
    def get_endpoint():
        return url_for('name.load', _external=True)

    live_server.start()

    res = client.post(
        get_endpoint(),
        headers={'Content-Type': 'application/json'},
        data={}
    )

    assert res.status_code == 200
```

```
def test_get_request(client, live_server):
    @live_server.app.route('/load-data')
    def get_endpoint():
```

(continues on next page)

(continued from previous page)

```
        return url_for('name.load', _external=True)

    live_server.start()

    res = client.get(get_endpoint())

    assert res.status_code == 200
```

Note: The notation `name.load_data`, corresponds to a `endpoint='load'` attribute, within a route decorator. The following is a route decorator using the blueprint implementation:

```
from flask import Blueprint, request

# local variables
blueprint = Blueprint(
    'name',
    __name__,
    template_folder='interface/templates',
    static_folder='interface/static'
)

@blueprint.route('/load-data', methods=['POST'], endpoint='load')
def load_data():
    if request.method == 'POST':
        if request.get_json():
            pass
```

Alternatively, the route function can be referenced directly from the `live_server` implementation, rather than implementing an `endpoint`:

```
def test_load_data(live_server, client):
    @live_server.app.route('/load-data', methods=['POST'])
    def load_data():
        pass

    live_server.start()

    res = client.post(url_for('load_data'), data={})
    assert res.status_code == 200
```

Note: Remember to explicitly define which `methods` are supported when registering the above route function.

2.2.1.7 Content negotiation

An important part of any REST (REpresentational State Transfer) service is content negotiation. It allows you to implement behaviour such as selecting a different serialization schemes for different media types.

> HTTP has provisions for several mechanisms for "content negotiation" - the process of selecting the best representation for a given response when there are multiple representations available.
>
> —RFC 2616#section-12. Fielding, et al.

The most common way to select one of the multiple possible representation is via `Accept` request header. The following series of `accept_*` fixtures provides an easy way to test content negotiation in your application:

```
def test_api_endpoint(accept_json, client):
    res = client.get(url_for('api.endpoint'), headers=accept_json)
    assert res.mimetype == 'application/json'
```

accept_any - */* accept header

`*/*` accept header suitable to use as parameter in `client`.

accept_json - application/json accept header

`application/json` accept header suitable to use as parameter in `client`.

accept_jsonp - application/json-p accept header

`application/json-p` accept header suitable to use as parameter in `client`.

2.2.2 Markers

`pytest-flask` registers the following markers. See the pytest documentation on what markers are and for notes on using them.

2.2.2.1 pytest.mark.options - pass options to your application config

pytest.mark.**options**(***kwargs*)

The mark used to pass options to your application config.

Parameters **kwargs** (*dict*) – The dictionary used to extend application config.

Example usage:

```
@pytest.mark.options(debug=False)
def test_app(app):
    assert not app.debug, 'Ensure the app is not in debug mode'
```

2.3 How to contribute

All contributions are greatly appreciated!

2.3.1 How to report issues

Facilitating the work of potential contributors is recommended since it increases the likelihood of your issue being solved quickly. The few extra steps listed below will help clarify problems you might be facing:

- Include a minimal reproducible example when possible.
- Describe the expected behaviour and what actually happened including a full trace-back in case of exceptions.

- Make sure to list details about your environment, such as your platform, versions of pytest, pytest-flask and python release.

Also, it's important to check the current open issues for similar reports in order to avoid duplicates.

2.3.2 Setting up your development environment

- Fork pytest-flask to your GitHub account by clicking the Fork button.
- Clone the main repository (not your fork) to your local machine.

```
$ git clone https://github.com/pytest-dev/pytest-flask
$ cd pytest-flask
```

- Add your fork as a remote to push your contributions.Replace `{username}` with your username.

```
$ git remote add fork https://github.com/{username}/pytest-flask
```

- Using Tox, create a virtual environment and install pytest-flask in editable mode with development dependencies.

```
$ tox -e dev
$ source venv/bin/activate
```

- Install pre-commit hooks

```
$ pre-commit install
```

2.3.3 Start Coding

- Create a new branch to identify what feature you are working on.

```
$ git fetch origin
$ git checkout -b your-branch-name origin/master
```

- Make your changes
- Include tests that cover any code changes you make and run them as described below.
- Push your changes to your fork. create a pull request describing your changes.

```
$ git push --set-upstream fork your-branch-name
```

2.3.4 How to run tests

You can run the test suite for the current environment with

```
$ pytest
```

To run the full test suite for all supported python versions

```
$ tox
```

Obs. CI will run tox when you submit your pull request, so this is optional.

2.3.5 Checking Test Coverage

To get a complete report of code sections not being touched by the test suite run `pytest` using `coverage`.

```
$ coverage run --concurrency=multiprocessing -m pytest
$ coverage combine
$ coverage html
```

Open `htmlcov/index.html` in your browser.

More about coverage here.

2.4 Changelog

2.4.1 1.2.1

- Fix bug in `:meth:pytest_flask.fixtures.live_server` where `SESSION_COOKIE_DOMAIN` was set to false due to `original_server_name` defaulting to "localhost". The new default is "localhost.localdomain".
- Drop support for python 3.6 and 3.5

2.4.2 1.2.0 (2021-02-26)

- Remove deprecated `:meth:live_server.url`
- fixture `request_ctx is now deprecated` and will be removed in the future
- `JSONReponse.json` removed in favour of `Werkzeug.wrappers.Response.json`

2.4.3 1.1.0 (2020-11-08)

- Speedup live server start time. Use *socket* instead of server pulling (#58) to check server availability and add new `--live-server-wait` option to set the live server wait timeout. Thanks to @jadkik.

2.4.4 1.0.0 (2020-03-03)

Important

- `live_server` is now `session`-scoped by default. This can be changed by using the `live-server_scope` option in your `pytest.ini` (#113). Thanks @havok2063 for the initial patch and @TWood67 for finishing it up.
- pytest 5.2 or later is now required.
- Python 2.7 and 3.4 are no longer supported.

2.4.5 0.15.1 (2020-02-03)

- Fix `ImportError` with `Werkzeug 1.0.0rc1` (#105).

2.4.6 0.15.0 (2019-05-13)

- Properly register the `options` marker (#97).

2.4.7 0.14.0 (2018-10-15)

- New `--live-server-host` command-line option to set the host name used by the `live_server` fixture.

  Thanks @o1da for the PR (#90).

2.4.8 0.13.0 (2018-09-29)

- `JSONReponse` now supports comparison directly with status codes:

```
assert client.get('invalid-route', headers=[('Accept', 'application/json')]) ==␣
↪404
```

  Thanks @dusktreader for the PR (#86).

2.4.9 0.12.0 (2018-09-06)

- `pytest-flask` now requires `pytest>=3.6` (#84).
- Add new `--live-server-port` option to select the port the live server will use (#82). Thanks @RazerM for the PR.
- Now `live_server` will try to stop the server cleanly by emitting a `SIGINT` signal and waiting 5 seconds for the server to shutdown. If the server is still running after 5 seconds, it will be forcefully terminated. This behavior can be changed by passing `--no-live-server-clean-stop` in the command-line (#49). Thanks @jadkik for the PR.
- Internal fixes silence pytest warnings, more visible now with `pytest-3.8.0` (#84).

2.4.10 0.11.0 (compared to 0.10.0)

- Implement deployment using Travis, following in line with many other pytest plugins.
- Allow live server to handle concurrent requests (#56), thanks to @mattwbarry for the PR.
- Fix broken link to pytest documentation (#50), thanks to @jineshpaloor for the PR.
- Tox support (#48), thanks to @steenzout for the PR.
- Add `LICENSE` into distribution (#43), thanks to @danstender.
- Minor typography improvements in documentation.
- Add changelog to documentation.

2.4.11 0.10.0 (compared to 0.9.0)

- Add `--start-live-server`/`--no-start-live-server` options to prevent live server from starting automatically (#36), thanks to @EliRibble.
- Fix title formatting in documentation.

2.4.12 0.9.0 (compared to 0.8.1)

- Rename marker used to pass options to application, e.g. `pytest.mark.app` is now `pytest.mark.options` (#35).
- Documentation badge points to the package documentation.
- Add Travis CI configuration to ensure the tests are passed in supported environments (#32).

2.4.13 0.8.1

- Minor changes in documentation.

2.4.14 0.8.0

- New `request_ctx` fixture which contains all request relevant information (#29).

2.4.15 0.7.5

- Use pytest `monkeypath` fixture to teardown application config (#27).

2.4.16 0.7.4

- Better test coverage, e.g. tests for available fixtures and markers.

2.4.17 0.7.3

- Use retina-ready badges in documentation (#21).

2.4.18 0.7.2

- Use pytest `monkeypatch` fixture to rewrite live server name.

2.4.19 0.7.1

- Single-sourcing package version (#24), as per "Python Packaging User Guide".

2.4.20 0.7.0

- Add package documentation (#20).

2.4.21 0.6.3

- Better documentation in README with reST formatting (#18), thanks to @greedo.

2.4.22 0.6.2

- Release the random port before starting the application live server (#17), thanks to @davehunt.

2.4.23 0.6.1

- Bind live server to a random port instead of 5000 or whatever is passed on the command line, so it's possible to execute tests in parallel via pytest-dev/pytest-xdist (#15). Thanks to @davehunt.
- Remove `--liveserver-port` option.

2.4.24 0.6.0

- Fix typo in option help for `--liveserver-port`, thanks to @svenstaro.

2.4.25 0.5.0

- Add `live_server` fixture uses to run application in the background (#11), thanks to @svenstaro.

2.4.26 0.4.0

- Add `client_class` fixture for class-based tests.

2.4.27 0.3.4

- Include package requirements into distribution (#8).

2.4.28 0.3.3

- Explicitly pin package dependencies and their versions.

2.4.29 0.3.2

- Use `codecs` module to open files to prevent possible errors on open files which contains non-ascii characters.

2.4.30 0.3.1

First release on PyPI.

Index

P

R