pytest-flask Documentation

Release 0.10.0

Vital Kudzelka

Oct 05, 2017

Contents

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

CHAPTER 1

User's Guide

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

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.

Step 1. Install

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

```
pip install pytest-flask
```

Step 2. Configure

Define your application fixture in `conftest.py`:

```
from myapp import create_app

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

Step 3. Run your test suite

Use the `py.test` command to run your test suite:

```
py.test
```

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.

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 *Contributing* section.

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 does not overrides. 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):
        '''What is the meaning of life, the universe and everything?'''
        return 42

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

```
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:

```
py.test -n <number of processes>
```

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

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.

`client` - application test client

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

Hint: During tests execution the request context has been pushed, e.g. `url_for`, `session` and other context bound objects are available without context managers.

Example:

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

`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
```

`config` - application config

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

`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 is starting 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: Your **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()
```

`request_ctx` - request context

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'
```

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 header suitable to use as parameter in `client`.

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

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

Markers

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

`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 not in debug mode'
```

Contributing

Found a bug? Have an issue?

The fastest way to get feedback on contributions/bugs is create a GitHub issue or catch me on Twitter.

Code syntax and conventions

We try to conform to **PEP 8** as much as possible. Make sure that your code as well.

Where are the tests?

Good that you're asking. The repository test suite is located in `tests` directory. Makefile defines a target to run them:

```
make test
```

Ensure the all tests are passed before submitting a pull request.

Changelog

Upcoming release

- 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.

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.

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).

0.8.1

- Minor changes in documentation.

0.8.0

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

0.7.5

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

0.7.4

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

0.7.3

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

0.7.2

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

0.7.1

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

0.7.0

- Add package documentation (#20).

0.6.3

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

0.6.2

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

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.

0.6.0

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

0.5.0

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

0.4.0

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

0.3.4

- Include package requirements into distribution (#8).

0.3.3

- Explicitly pin package dependencies and their versions.

0.3.2

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

0.3.1

First release on PyPI.

CHAPTER 2

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
```

And run your test suite:

```
py.test
```

CHAPTER 3

Contributing

Don't hesitate to create a GitHub issue for any **bug** or **suggestion**.

Index