# pytest-sanic Documentation

***Release stable***

**Oct 25, 2021**

# Contents

A pytest plugin for Sanic. It helps you to test your code asynchronously.

This plugin provides:

- very easy testing with async coroutines
- common and useful fixtures
- asynchronous fixture support
- test_client/sanic_client for Sanic application
- test_server for Sanic application

# CHAPTER 1

## Quick Start

You don't have to load `pytest-sanic` explicitly. `pytest` will do it for you. Just write tests like,

```
async def test_sanic_db_find_by_id(app):
    """
    Let's assume that, in db we have,
        {
            "id": "123",
            "name": "Kobe Bryant",
            "team": "Lakers",
        }
    """
    doc = await app.db["players"].find_by_id("123")
    assert doc.name == "Kobe Bryant"
    assert doc.team == "Lakers"
```

Here's much more realistic & useful example,

```
from .app import create_app

@pytest.yield_fixture
def app():
    app = create_app(test_config, **params)
    yield app

@pytest.fixture
def test_cli(loop, app, sanic_client):
    return loop.run_until_complete(sanic_client(app))

async def test_index(test_cli):
    resp = await test_cli.get('/')
    assert resp.status_code == 200

async def test_player(test_cli):
    resp = await test_cli.get('/player')
    assert resp.status_code == 200
```

Contents:

## 1.1 Installation

### 1.1.1 Installing it globally

You can install pytest-sanic globally with any Python package manager:

```
pip install pytest-sanic
```

## 1.2 Example

A simple example.

### 1.2.1 Quick Start

You don't have to load `pytest-sanic` explicitly. `pytest` will do it for you. Just write tests like,

```
async def test_sanic_db_find_by_id(app):
    """
    Let's assume that, in db we have,
        {
            "id": "123",
            "name": "Kobe Bryant",
            "team": "Lakers",
        }
    """
    doc = await app.db["players"].find_by_id("123")
    assert doc.name == "Kobe Bryant"
    assert doc.team == "Lakers"
```

## 1.3 Fixtures

Some fixtures for easy testing.

### 1.3.1 loop

`pytest-sanic` creates an event loop and injects it as a fixture. `pytest` will use this event loop to run your `async tests`. By default, fixture `loop` is an instance of *asyncio.new_event_loop*. But *uvloop* is also an option for you, by simpy passing `--loop uvloop`. Keep mind to just use one single event loop.

### 1.3.2 unused_port

an unused TCP port on the localhost.

### 1.3.3 test_server

Creates a TestServer instance by giving a `Sanic` application. It's very easy to utilize `test_server` to create your *Sanic* application server for testing.

```
@pytest.yield_fixture
def app():
    app = Sanic("test_sanic_app")

    @app.route("/test_get", methods=['GET'])
    async def test_get(request):
        return response.json({"GET": True})

    yield app

@pytest.fixture
def sanic_server(loop, app, test_server):
    return loop.run_until_complete(test_server(app))
```

You can also very easily override this `loop` fixture by creating your own, simply like,

```
@pytest.yield_fixture
def loop():
    loop = MyEventLoop()
    yield loop
    loop.close()
```

### 1.3.4 test_client

`test_client` has been deprecated, please use *sanic_client* instead, check out issue for more context.

### 1.3.5 sanic_client

Creates a TestClient instance by giving a `Sanic` application. You can simply have a client by using `sanic_client`, like

```
@pytest.yield_fixture
def app():
    app = Sanic("test_sanic_app")

    @app.route("/test_get", methods=['GET'])
    async def test_get(request):
        return response.json({"GET": True})

    @app.route("/test_post", methods=['POST'])
    async def test_post(request):
        return response.json({"POST": True})

    @app.route("/test_put", methods=['PUT'])
    async def test_put(request):
        return response.json({"PUT": True})

    @app.route("/test_delete", methods=['DELETE'])
    async def test_delete(request):
        return response.json({"DELETE": True})
```

(continues on next page)

(continued from previous page)

```
    @app.route("/test_patch", methods=['PATCH'])
    async def test_patch(request):
        return response.json({"PATCH": True})

    @app.route("/test_options", methods=['OPTIONS'])
    async def test_options(request):
        return response.json({"OPTIONS": True})

    @app.route("/test_head", methods=['HEAD'])
    async def test_head(request):
        return response.json({"HEAD": True})

    @app.websocket("/test_ws")
    async def test_ws(request, ws):
        data = await ws.recv()
        await ws.send(data)

    yield app

@pytest.fixture
def test_cli(loop, app, sanic_client):
    return loop.run_until_complete(sanic_client(app, protocol=WebSocketProtocol))

#########
# Tests #
#########

async def test_fixture_test_client_get(test_cli):
    """
    GET request
    """
    resp = await test_cli.get('/test_get')
    assert resp.status_code == 200
    resp_json = resp.json()
    assert resp_json == {"GET": True}

async def test_fixture_test_client_post(test_cli):
    """
    POST request
    """
    resp = await test_cli.post('/test_post')
    assert resp.status_code == 200
    resp_json = resp.json()
    assert resp_json == {"POST": True}

async def test_fixture_test_client_put(test_cli):
    """
    PUT request
    """
    resp = await test_cli.put('/test_put')
    assert resp.status_code == 200
    resp_json = resp.json()
    assert resp_json == {"PUT": True}

async def test_fixture_test_client_delete(test_cli):
    """
```

(continues on next page)

(continued from previous page)

```
    DELETE request
    """
    resp = await test_cli.delete('/test_delete')
    assert resp.status_code == 200
    resp_json = resp.json()
    assert resp_json == {"DELETE": True}

async def test_fixture_test_client_patch(test_cli):
    """
    PATCH request
    """
    resp = await test_cli.patch('/test_patch')
    assert resp.status_code == 200
    resp_json = resp.json()
    assert resp_json == {"PATCH": True}

async def test_fixture_test_client_options(test_cli):
    """
    OPTIONS request
    """
    resp = await test_cli.options('/test_options')
    assert resp.status_code == 200
    resp_json = resp.json()
    assert resp_json == {"OPTIONS": True}

async def test_fixture_test_client_head(test_cli):
    """
    HEAD request
    """
    resp = await test_cli.head('/test_head')
    assert resp.status_code == 200
    resp_json = resp.json()
    # HEAD should not have body
    assert resp_json is None

async def test_fixture_test_client_ws(test_cli):
    """
    Websockets
    """
    ws_conn = await test_cli.ws_connect('/test_ws')
    data = 'hello world!'
    await ws_conn.send(data)
    msg = await ws_conn.recv()
    assert msg == data
    await ws_conn.close()
```

small notes:

`test_cli.ws_connect` does not work in `sanic.__version__ <= '0.5.4'`, because of a Sanic bug, but it has been fixed in master branch. And `websockets.__version__ >= '4.0'` has broken websockets in `sanic.__version__ <= '0.6.0'`, but it has been fixed in master.

## 1.4 Q&A

### 1.4.1 Tips

- Blueprints Testing
- `test_cli.ws_connect` does not work in `sanic.__version__ <= '0.5.4'`, because of a Sanic bug, but it has been fixed in master branch.
- Importing app has loop already running when you have *db_init* listeners.
- Incorrect coverage report with `pytest-cov`, but we can have workarounds for this issue, it's a pytest loading plugin problem essentially.
- Websockets > 4.0 has broken websockets in `sanic.__version__ <= '0.6.0'`, but it has been fixed in master
- `sanic_client/test_client` is not an instance of `sanic.testing.SanicTestClient`. It is a simpler interface designed for easier testing. It randomizes the port used. The return value of `get`, `post`, etc is `Response` unlike `(Request, Response)` as that of `SanicTestClient`. See here for more info.

Feel free to create issue if you have any question. You can also check out closed issues

## 1.5 Development

### 1.5.1 Contribution

`pytest-sanic` accepts contributions on GitHub, in the form of issues or pull requests.

Build.

```
poetry install
```

Run unit tests.

```
poetry run pytest ./tests --cov pytest_sanic
```

## 1.6 Reference

Some useful pytest plugins:

- pytest-tornado
- pytest-asyncio
- pytest-aiohttp

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search