Sanic-cn Documentation

0.7.0

chinesejar

2018 05 12

Contents

1 **1**

2 Routing **3**
2.1 . . . 3
2.2 HTTP . . . 4
2.3 `add_route` . . . 5
2.4 URL `url_for` . . . 5
2.5 WebSocket . . . 6
2.6 `strict_slashes` . . . 6
2.7 . . . 7
2.8 URL . . . 8

3 **9**
3.1 `get getlist` . . . 11

4 **13**
4.1 . . . 13
4.2 HTML . . . 13
4.3 JSON . . . 13
4.4 . . . 14
4.5 . . . 14
4.6 . . . 14
4.7 . . . 14
4.8 Raw . . . 14
4.9 . . . 15

5 **17**

6 **19**
6.1 . . . 19
6.2 . . . 19
6.3 . . . 20

7 **21**
7.1 . . . 21
7.2 . . . 21
7.3 . . . 22

7.4 . . . 22

8 **25**
8.1 . . . 25
8.2 . . . 25
8.3 . . . 26
8.4 . . . 27
8.5 . . . 27
8.6 : API . . . 28
8.7 `url_for` URL . . . 28

9 WebSocket **31**

10 Configuration **33**
10.1 Basics . . . 33
10.2 Loading Configuration . . . 33
10.3 Builtin Configuration Values . . . 34

11 Cookies **37**
11.1 Reading cookies . . . 37
11.2 Writing cookies . . . 37
11.3 Deleting cookies . . . 38

12 Handler Decorators **39**
12.1 Authorization Decorator . . . 39

13 Streaming **41**
13.1 Request Streaming . . . 41
13.2 Response Streaming . . . 42

14 Class-Based Views **45**
14.1 Defining views . . . 45
14.2 URL parameters . . . 46
14.3 Decorators . . . 46
14.4 URL Building . . . 47
14.5 Using CompositionView . . . 47

15 Custom Protocols **49**
15.1 Example . . . 49

16 SSL Example **51**

17 Logging **53**
17.1 Quick Start . . . 53
17.2 Configuration . . . 54

18 Testing **55**
18.1 pytest-sanic . . . 56

19 Deploying **59**
19.1 Workers . . . 59
19.2 Running via command . . . 60
19.3 Running via Gunicorn . . . 60
19.4 Asynchronous support . . . 60

20 Extensions **61**

21 Contributing **63**
21.1 Installation . . . 63
21.2 Running tests . . . 63
21.3 Pull requests! . . . 63
21.4 Documentation . . . 64
21.5 Warning . . . 64

22 API Reference **65**
22.1 Submodules . . . 66
22.2 sanic.app module . . . 66
22.3 sanic.blueprints module . . . 66
22.4 sanic.config module . . . 66
22.5 sanic.constants module . . . 66
22.6 sanic.cookies module . . . 66
22.7 sanic.exceptions module . . . 66
22.8 sanic.handlers module . . . 66
22.9 sanic.log module . . . 66
22.10 sanic.request module . . . 66
22.11 sanic.response module . . . 66
22.12 sanic.router module . . . 66
22.13 sanic.server module . . . 66
22.14 sanic.static module . . . 66
22.15 sanic.testing module . . . 66
22.16 sanic.views module . . . 66
22.17 sanic.websocket module . . . 66
22.18 sanic.worker module . . . 66
22.19 Module contents . . . 66

23 Indices and tables **67**

CHAPTER 1

pip 3.5 Python Sanic `async/await`

1. Sanic: `python3 -m pip install sanic`
2. `main.py`

```
from sanic import Sanic
from sanic.response import json

app = Sanic()

@app.route("/")
async def test(request):
    return json({"hello": "world"})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

1. : `python3 main.py`
2. `http://0.0.0.0:8000` *Hello world!*

Sanic

CHAPTER 2

Routing

URL

app Sanic

```
from sanic.response import json

@app.route("/")
async def test(request):
    return json({ "hello": "world" })
```

http://server.url/ url (url), / test JSON

Sanic async def

2.1

Sanic

<PARAM>

```
from sanic.response import text

@app.route('/tag/<tag>')
async def tag_handler(request, tag):
    return text('Tag - {}'.format(tag))
```

:type Sanic NotFound URL 404: Page not found

```
from sanic.response import text

@app.route('/number/<integer_arg:int>')
async def integer_handler(request, integer_arg):
    return text('Integer - {}'.format(integer_arg))
```

(continues on next page)

()

```
@app.route('/number/<number_arg:number>')
async def number_handler(request, number_arg):
    return text('Number - {}'.format(number_arg))

@app.route('/person/<name:[A-z]+>')
async def person_handler(request, name):
    return text('Person - {}'.format(name))

@app.route('/folder/<folder_id:[A-z0-9]{0,4}>')
async def folder_handler(request, folder_id):
    return text('Folder - {}'.format(folder_id))
```

2.2 HTTP

URL GET `@app.route` `methods` HTTP

```
from sanic.response import text

@app.route('/post', methods=['POST'])
async def post_handler(request):
    return text('POST request - {}'.format(request.json))

@app.route('/get', methods=['GET'])
async def get_handler(request):
    return text('GET request - {}'.format(request.args))
```

`host` (list string) host

```
@app.route('/get', methods=['GET'], host='example.com')
async def get_handler(request):
    return text('GET request - {}'.format(request.args))

# if the host header doesn't match example.com, this route will be used
@app.route('/get', methods=['GET'])
async def get_handler(request):
    return text('GET request in default - {}'.format(request.args))
```

```
from sanic.response import text

@app.post('/post')
async def post_handler(request):
    return text('POST request - {}'.format(request.json))

@app.get('/get')
async def get_handler(request):
    return text('GET request - {}'.format(request.args))
```

2.3 add_route

@app.route app.add_route

```
from sanic.response import text

# Define the handler functions
async def handler1(request):
    return text('OK')

async def handler2(request, name):
    return text('Folder - {}'.format(name))

async def person_handler2(request, name):
    return text('Person - {}'.format(name))

# Add each handler function as a route
app.add_route(handler1, '/test')
app.add_route(handler2, '/folder/<name>')
app.add_route(person_handler2, '/person/<name:[A-z]>', methods=['GET'])
```

2.4 URL url_for

Sanic url_for URL app url

```
from sanic.response import redirect

@app.route('/')
async def index(request):
    # generate a URL for the endpoint `post_handler`
    url = app.url_for('post_handler', post_id=5)
    # the URL is `/posts/5`, redirect to it
    return redirect(url)

@app.route('/posts/<post_id>')
async def post_handler(request, post_id):
    return text('Post - {}'.format(post_id))
```

url_for

- url_for URL

```
url = app.url_for('post_handler', post_id=5, arg_one='one', arg_two='two')
# /posts/5?arg_one=one&arg_two=two
```

- url_for

```
url = app.url_for('post_handler', post_id=5, arg_one=['one', 'two'])
# /posts/5?arg_one=one&arg_one=two
```

- (_anchor, _external, _scheme, _method, _server) url_for url (_method)

```
url = app.url_for('post_handler', post_id=5, arg_one='one', _anchor='anchor')
# /posts/5?arg_one=one#anchor
```

(continues on next page)

()

```
url = app.url_for('post_handler', post_id=5, arg_one='one', _external=True)
# //server/posts/5?arg_one=one
# _external requires passed argument _server or SERVER_NAME in app.config or url will␣
↪be same as no _external

url = app.url_for('post_handler', post_id=5, arg_one='one', _scheme='http', _
↪external=True)
# http://server/posts/5?arg_one=one
# when specifying _scheme, _external must be True

# you can pass all special arguments one time
url = app.url_for('post_handler', post_id=5, arg_one=['one', 'two'], arg_two=2, _
↪anchor='anchor', _scheme='http', _external=True, _server='another_server:8888')
# http://another_server:8888/posts/5?arg_one=one&arg_one=two&arg_two=2#anchor
```

- `url_for` URL `URLBuildError`

2.5 WebSocket

WebSocket `@app.websocket`

```
@app.websocket('/feed')
async def feed(request, ws):
    while True:
        data = 'hello!'
        print('Sending: ' + data)
        await ws.send(data)
        data = await ws.recv()
        print('Received: ' + data)
```

,`app.add_websocket_route`

```
async def feed(request, ws):
    pass

app.add_websocket_route(my_websocket_handler, '/feed')
```

WebSocket WebSocket `send` `recv`

WebSocket Aymeric Augustin websockets

2.6 strict_slashes

`routes`

```


# provide default strict_slashes value for all routes
app = Sanic('test_route_strict_slash', strict_slashes=True)

# you can also overwrite strict_slashes value for specific route
@app.get('/get', strict_slashes=False)
def handler(request):
    return text('OK')
```

(continues on next page)

()

```
# It also works for blueprints
bp = Blueprint('test_bp_strict_slash', strict_slashes=True)

@bp.get('/bp/get', strict_slashes=False)
def handler(request):
    return text('OK')

app.blueprint(bp)
```

2.7

name (handler.__name__)

```
app = Sanic('test_named_route')

@app.get('/get', name='get_handler')
def handler(request):
    return text('OK')

# then you need use `app.url_for('get_handler')`
# instead of # `app.url_for('handler')`

# It also works for blueprints
bp = Blueprint('test_named_bp')

@bp.get('/bp/get', name='get_handler')
def handler(request):
    return text('OK')

app.blueprint(bp)

# then you need use `app.url_for('test_named_bp.get_handler')`
# instead of `app.url_for('test_named_bp.handler')`

# different names can be used for same url with different methods

@app.get('/test', name='route_test')
def handler(request):
    return text('OK')

@app.post('/test', name='route_post')
def handler2(request):
    return text('OK POST')

@app.put('/test', name='route_put')
def handler3(request):
    return text('OK PUT')

# below url are the same, you can use any of them
# '/test'
app.url_for('route_test')
# app.url_for('route_post')
```

(continues on next page)

()

```
# app.url_for('route_put')

# for same handler name with different methods
# you need specify the name (it's url_for issue)
@app.get('/get')
def handler(request):
    return text('OK')

@app.post('/post', name='post_handler')
def handler(request):
    return text('OK')

# then
# app.url_for('handler') == '/get'
# app.url_for('post_handler') == '/post'
```

2.8 URL

url_for url filename

```
app = Sanic('test_static')
app.static('/static', './static')
app.static('/uploads', './uploads', name='uploads')
app.static('/the_best.png', '/home/ubuntu/test.png', name='best_png')

bp = Blueprint('bp', url_prefix='bp')
bp.static('/static', './static')
bp.static('/uploads', './uploads', name='uploads')
bp.static('/the_best.png', '/home/ubuntu/test.png', name='best_png')
app.blueprint(bp)

# then build the url
app.url_for('static', filename='file.txt') == '/static/file.txt'
app.url_for('static', name='static', filename='file.txt') == '/static/file.txt'
app.url_for('static', name='uploads', filename='file.txt') == '/uploads/file.txt'
app.url_for('static', name='best_png') == '/the_best.png'

# blueprint url building
app.url_for('static', name='bp.static', filename='file.txt') == '/bp/static/file.txt'
app.url_for('static', name='bp.uploads', filename='file.txt') == '/bp/uploads/file.txt
↪'
app.url_for('static', name='bp.best_png') == '/bp/static/the_best.png'
```

CHAPTER 3

HTTP `Request`

`Request`

- `json` () - JSON body

```
from sanic.response import json

@app.route("/json")
def post_json(request):
    return json({ "received": True, "message": request.json })
```

- `args` () - `?key1=value1&key2=value2` URL `args` `{'key1': ['value1'], 'key2': ['value2']}` `query_string`

```
from sanic.response import json

@app.route("/query_string")
def query_string(request):
    return json({ "parsed": True, "args": request.args, "url": request.url,
↪"query_string": request.query_string })
```

- `raw_args` () - url URL `?key1=value1&key2=value2` `raw_args` `{'key1': 'value1', 'key2': 'value2'}`
- `files` (`File`) - name, body, type

```
from sanic.response import json

@app.route("/files")
def post_json(request):
    test_file = request.files.get('test')

    file_parameters = {
        'body': test_file.body,
        'name': test_file.name,
        'type': test_file.type,
```

(continues on next page)

()

```
    }

    return json({ "received": True, "file_names": request.files.keys(), "test_
↪file_parameters": file_parameters })
```

- form () - form

```
from sanic.response import json

@app.route("/form")
def post_json(request):
    return json({ "received": True, "form_data": request.form, "test": request.
↪form.get('test') })
```

- body (bytes) -

```
from sanic.response import text

@app.route("/users", methods=["POST",])
def create_user(request):
    return text("You are trying to create a user with the following POST: %s" %␣
↪request.body)
```

- headers () - -
- method () - HTTP (ie GET, POST)
- ip () - IP
- port () -
- socket () - (IP, port).
- app - Sanic app

```
from sanic.response import json
from sanic import Blueprint

bp = Blueprint('my_blueprint')

@bp.route('/')
async def bp_root(request):
    if request.app.config['DEBUG']:
        return json({'status': 'debug'})
    else:
        return json({'status': 'production'})
```

- url: URL ie: http://localhost:8000/posts/1/?foo=bar
- scheme: URL : http https
- host: : localhost:8080
- path: : /posts/1/
- query_string: : foo=bar ''
- uri_template: /posts/<id>/
- token: : Basic YWRtaW46YWRtaW4=

3.1 get getlist

RequestParameters dict get getlist

- get(key, default=None) key
- getlist(key, default=None).

```
from sanic.request import RequestParameters

args = RequestParameters()
args['titles'] = ['Post 1', 'Post 2']

args.get('titles') # => 'Post 1'

args.getlist('titles') # => ['Post 1', 'Post 2']
```

CHAPTER 4

sanic.response

4.1

```
from sanic import response


@app.route('/text')
def handle_request(request):
    return response.text('Hello world!')
```

4.2 HTML

```
from sanic import response


@app.route('/html')
def handle_request(request):
    return response.html('<p>Hello world!</p>')
```

4.3 JSON

```
from sanic import response


@app.route('/json')
def handle_request(request):
    return response.json({'message': 'Hello world!'})
```

4.4

```
from sanic import response


@app.route('/file')
async def handle_request(request):
    return await response.file('/srv/www/whatever.png')
```

4.5

```
from sanic import response

@app.route("/streaming")
async def index(request):
    async def streaming_fn(response):
        response.write('foo')
        response.write('bar')
    return response.stream(streaming_fn, content_type='text/plain')
```

4.6

```
from sanic import response

@app.route('/big_file.png')
async def handle_request(request):
    return await response.file_stream('/srv/www/whatever.png')
```

4.7

```
from sanic import response


@app.route('/redirect')
def handle_request(request):
    return response.redirect('/json')
```

4.8 Raw

body

```
from sanic import response


@app.route('/raw')
```

(continues on next page)

()

```
def handle_request(request):
    return response.raw(b'raw data')
```

4.9

headers status

```
from sanic import response


@app.route('/json')
def handle_request(request):
    return response.json(
        {'message': 'Hello world!'},
        headers={'X-Served-By': 'sanic'},
        status=200
    )
```

CHAPTER 5

app.static URL

```
from sanic import Sanic
from sanic.blueprints import Blueprint

app = Sanic(__name__)

# Serves files from the static folder to the URL /static
app.static('/static', './static')
# use url_for to build the url, name defaults to 'static' and can be ignored
app.url_for('static', filename='file.txt') == '/static/file.txt'
app.url_for('static', name='static', filename='file.txt') == '/static/file.txt'

# Serves the file /home/ubuntu/test.png when the URL /the_best.png
# is requested
app.static('/the_best.png', '/home/ubuntu/test.png', name='best_png')

# you can use url_for to build the static file url
# you can ignore name and filename parameters if you don't define it
app.url_for('static', name='best_png') == '/the_best.png'
app.url_for('static', name='best_png', filename='any') == '/the_best.png'

# you need define the name for other static files
app.static('/another.png', '/home/ubuntu/another.png', name='another')
app.url_for('static', name='another') == '/another.png'
app.url_for('static', name='another', filename='any') == '/another.png'

# also, you can use static for blueprint
bp = Blueprint('bp', url_prefix='/bp')
bp.static('/static', './static')

# servers the file directly
bp.static('/the_best.png', '/home/ubuntu/test.png', name='best_png')
app.blueprint(bp)

app.url_for('static', name='bp.static', filename='file.txt') == '/bp/static/file.txt'
```

(continues on next page)

()

```
app.url_for('static', name='bp.best_png') == '/bp/test_best.png'

app.run(host="0.0.0.0", port=8000)
```

CHAPTER 6

Sanic HTTP

6.1

sanic.exceptions raise

```
from sanic.exceptions import ServerError

@app.route('/killme')
async def i_am_ready_to_die(request):
    raise ServerError("Something bad happened", status_code=500)
```

abort

```
from sanic.exceptions import abort
from sanic.response import text

@app.route('/youshallnotpass')
async def no_no(request):
        abort(401)
        # this won't happen
        text("OK")
```

6.2

Sanic @app.exception SanicException Request Exception

```
from sanic.response import text
from sanic.exceptions import NotFound

@app.exception(NotFound)
```

(continues on next page)

()

```
async def ignore_404s(request, exception):
    return text("Yep, I totally found the page: {}".format(request.url))
```

6.3

- NotFound:
- ServerError:

sanic.exceptions

CHAPTER 7

Sanic

7.1

@app.middleware 'request' 'response'.

- request
- request response

```
@app.middleware('request')
async def print_on_request(request):
    print("I print when a request is received by the server")

@app.middleware('response')
async def print_on_response(request, response):
    print("I print when a response is returned by the server")
```

7.2

```
app = Sanic(__name__)

@app.middleware('response')
async def custom_banner(request, response):
    response.headers["Server"] = "Fake-Server"

@app.middleware('response')
async def prevent_xss(request, response):
    response.headers["x-xss-protection"] = "1; mode=block"

app.run(host="0.0.0.0", port=8000)
```

custom_banner HTTP *Server* *Fake-Server* **prevent_xss** HTTP (XSS)

7.3

`HTTPResponse`

```
@app.middleware('request')
async def halt_request(request):
    return text('I halted the request')

@app.middleware('response')
async def halt_response(request, response):
    return text('I halted the response')
```

7.4

startup/teardown

- `before_server_start`
- `after_server_start`
- `before_server_stop`
- `after_server_stop`

app asyncio loop

```
@app.listener('before_server_start')
async def setup_db(app, loop):
    app.db = await db_setup()

@app.listener('after_server_start')
async def notify_server_started(app, loop):
    print('Server successfully started!')

@app.listener('before_server_stop')
async def notify_server_stopping(app, loop):
    print('Server shutting down!')

@app.listener('after_server_stop')
async def close_db(app, loop):
    await app.db.close()
```

`register_listener` app

```
app = Sanic()

async def setup_db(app, loop):
    app.db = await db_setup()

app.register_listener(setup_db, 'before_server_start')
```

loop Sanic `add_task`

```
async def notify_server_started_after_five_seconds():
    await asyncio.sleep(5)
    print('Server successfully started!')

app.add_task(notify_server_started_after_five_seconds())
```

Sanic app

```
async def notify_server_started_after_five_seconds(app):
    await asyncio.sleep(5)
    print(app.name)

app.add_task(notify_server_started_after_five_seconds)
```

app

```
async def notify_server_started_after_five_seconds(app):
    await asyncio.sleep(5)
    print(app.name)

app.add_task(notify_server_started_after_five_seconds(app))
```

CHAPTER 8

(Blueprints)

8.1

url /

import `my_blueprint.py`

```
from sanic.response import json
from sanic import Blueprint

bp = Blueprint('my_blueprint')

@bp.route('/')
async def bp_root(request):
    return json({'my': 'blueprint'})
```

8.2

```
from sanic import Sanic
from my_blueprint import bp

app = Sanic(__name__)
app.blueprint(bp)

app.run(host='0.0.0.0', port=8000, debug=True)
```

`app.router`

```
[Route(handler=<function bp_root at 0x7f908382f9d8>, methods=None, pattern=re.compile(
↪'^/$'), parameters=[])]
```

8.3

Blueprint.group ’’ ()

```
api/
├──content/
│   ├──authors.py
│   ├──static.py
│   └──__init__.py
├──info.py
└──__init__.py
app.py
```

```
# api/content/authors.py
from sanic import Blueprint

authors = Blueprint('content_authors', url_prefix='/authors')
```

```
# api/content/static.py
from sanic import Blueprint

static = Blueprint('content_static', url_prefix='/static')
```

```
# api/content/__init__.py
from sanic import Blueprint

from .static import static
from .authors import authors

content = Blueprint.group(assets, authors, url_prefix='/content')
```

```
# api/info.py
from sanic import Blueprint

info = Blueprint('info', url_prefix='/info')
```

```
# api/__init__.py
from sanic import Blueprint

from .content import content
from .info import info

api = Blueprint.group(content, info, url_prefix='/api')
```

app.py

```
# app.py
from sanic import Sanic

from .api import api

app = Sanic(__name__)

app.blueprint(api)
```

8.4

8.4.1 WebSocket

WebSocket `@bp.websocket` `bp.add_websocket_route`

8.4.2

```
@bp.middleware
async def print_on_request(request):
    print("I am a spy")

@bp.middleware('request')
async def halt_request(request):
    return text('I halted the request')

@bp.middleware('response')
async def halt_response(request, response):
    return text('I halted the response')
```

8.4.3

```
@bp.exception(NotFound)
def ignore_404s(request, exception):
    return text("Yep, I totally found the page: {}".format(request.url))
```

8.4.4

```
# suppose bp.name == 'bp'

bp.static('/web/path', '/folder/to/serve')
# also you can pass name parameter to it for url_for
bp.static('/web/path', '/folder/to/server', name='uploads')
app.url_for('static', name='bp.uploads', filename='file.txt') == '/bp/web/path/file.
↪txt'
```

8.5

(worker) workers fork

:

- `before_server_start`:
- `after_server_start`:
- `before_server_stop`:
- `after_server_stop`:

```
bp = Blueprint('my_blueprint')

@bp.listener('before_server_start')
async def setup_connection(app, loop):
    global database
    database = mysql.connect(host='127.0.0.1'...)

@bp.listener('after_server_stop')
async def close_connection(app, loop):
    await database.close()
```

8.6 : API

API /v1/<routes> /v2/<routes>

url_prefix API

```
# blueprints.py
from sanic.response import text
from sanic import Blueprint

blueprint_v1 = Blueprint('v1', url_prefix='/v1')
blueprint_v2 = Blueprint('v2', url_prefix='/v2')

@blueprint_v1.route('/')
async def api_v1_root(request):
    return text('Welcome to version 1 of our documentation')

@blueprint_v2.route('/')
async def api_v2_root(request):
    return text('Welcome to version 2 of our documentation')
```

app /v1 /v2 API

```
# main.py
from sanic import Sanic
from blueprints import blueprint_v1, blueprint_v2

app = Sanic(__name__)
app.blueprint(blueprint_v1, url_prefix='/v1')
app.blueprint(blueprint_v2, url_prefix='/v2')

app.run(host='0.0.0.0', port=8000, debug=True)
```

8.7 url_for URL

URL <blueprint_name>.<handler_name>

```
@blueprint_v1.route('/')
async def root(request):
    url = request.app.url_for('v1.post_handler', post_id=5) # --> '/v1/post/5'
    return redirect(url)
```

(continues on next page)

()

```
@blueprint_v1.route('/post/<post_id>')
async def post_handler(request, post_id):
    return text('Post {} in Blueprint V1'.format(post_id))
```

CHAPTER 9

WebSocket

Sanic supports websockets, to setup a WebSocket:

```
from sanic import Sanic
from sanic.response import json
from sanic.websocket import WebSocketProtocol

app = Sanic()

@app.websocket('/feed')
async def feed(request, ws):
    while True:
        data = 'hello!'
        print('Sending: ' + data)
        await ws.send(data)
        data = await ws.recv()
        print('Received: ' + data)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, protocol=WebSocketProtocol)
```

Alternatively, the `app.add_websocket_route` method can be used instead of the decorator:

```
async def feed(request, ws):
    pass

app.add_websocket_route(feed, '/feed')
```

Handlers for a WebSocket route are passed the request as first argument, and a WebSocket protocol object as second argument. The protocol object has `send` and `recv` methods to send and receive data respectively.

You could setup your own WebSocket configuration through `app.config`, like

Find more in `Configuration` section.

CHAPTER 10

Configuration

Any reasonably complex application will need configuration that is not baked into the actual code. Settings might be different for different environments or installations.

10.1 Basics

Sanic holds the configuration in the `config` attribute of the application object. The configuration object is merely an object that can be modified either using dot-notation or like a dictionary:

```
app = Sanic('myapp')
app.config.DB_NAME = 'appdb'
app.config.DB_USER = 'appuser'
```

Since the config object actually is a dictionary, you can use its `update` method in order to set several values at once:

```
db_settings = {
    'DB_HOST': 'localhost',
    'DB_NAME': 'appdb',
    'DB_USER': 'appuser'
}
app.config.update(db_settings)
```

In general the convention is to only have UPPERCASE configuration parameters. The methods described below for loading configuration only look for such uppercase parameters.

10.2 Loading Configuration

There are several ways how to load configuration.

10.2.1 From Environment Variables

Any variables defined with the `SANIC_` prefix will be applied to the sanic config. For example, setting `SANIC_REQUEST_TIMEOUT` will be loaded by the application automatically and fed into the `REQUEST_TIMEOUT` config variable. You can pass a different prefix to Sanic:

```
app = Sanic(load_env='MYAPP_')
```

Then the above variable would be `MYAPP_REQUEST_TIMEOUT`. If you want to disable loading from environment variables you can set it to `False` instead:

```
app = Sanic(load_env=False)
```

10.2.2 From an Object

If there are a lot of configuration values and they have sensible defaults it might be helpful to put them into a module:

```
import myapp.default_settings

app = Sanic('myapp')
app.config.from_object(myapp.default_settings)
```

You could use a class or any other object as well.

10.2.3 From a File

Usually you will want to load configuration from a file that is not part of the distributed application. You can load configuration from a file using `from_pyfile(/path/to/config_file)`. However, that requires the program to know the path to the config file. So instead you can specify the location of the config file in an environment variable and tell Sanic to use that to find the config file:

```
app = Sanic('myapp')
app.config.from_envvar('MYAPP_SETTINGS')
```

Then you can run your application with the `MYAPP_SETTINGS` environment variable set:

```
$ MYAPP_SETTINGS=/path/to/config_file python3 myapp.py
INFO: Goin' Fast @ http://0.0.0.0:8000
```

The config files are regular Python files which are executed in order to load them. This allows you to use arbitrary logic for constructing the right configuration. Only uppercase variables are added to the configuration. Most commonly the configuration consists of simple key value pairs:

```
# config_file
DB_HOST = 'localhost'
DB_NAME = 'appdb'
DB_USER = 'appuser'
```

10.3 Builtin Configuration Values

Out of the box there are just a few predefined values which can be overwritten when creating the application.

Variable	Default	Description
REQUEST_MAX_SIZE	100000000	How big a request may be (bytes)
REQUEST_TIMEOUT	60	How long a request can take to arrive (sec)
RESPONSE_TIMEOUT	60	How long a response can take to process (sec)
KEEP_ALIVE	**True**	Disables keep-alive when **False**
KEEP_ALIVE_TIMEOUT	5	How long to hold a TCP connection open (sec)

10.3.1 The different Timeout variables:

A request timeout measures the duration of time between the instant when a new open TCP connection is passed to the Sanic backend server, and the instant when the whole HTTP request is received. If the time taken exceeds the `REQUEST_TIMEOUT` value (in seconds), this is considered a Client Error so Sanic generates a HTTP 408 response and sends that to the client. Adjust this value higher if your clients routinely pass very large request payloads or upload requests very slowly.

A response timeout measures the duration of time between the instant the Sanic server passes the HTTP request to the Sanic App, and the instant a HTTP response is sent to the client. If the time taken exceeds the `RESPONSE_TIMEOUT` value (in seconds), this is considered a Server Error so Sanic generates a HTTP 503 response and sets that to the client. Adjust this value higher if your application is likely to have long-running process that delay the generation of a response.

10.3.2 What is Keep Alive? And what does the Keep Alive Timeout value do?

Keep-Alive is a HTTP feature indroduced in HTTP 1.1. When sending a HTTP request, the client (usually a web browser application) can set a Keep-Alive header to indicate for the http server (Sanic) to not close the TCP connection after it has send the response. This allows the client to reuse the existing TCP connection to send subsequent HTTP requests, and ensures more efficient network traffic for both the client and the server.

The `KEEP_ALIVE` config variable is set to `True` in Sanic by default. If you don't need this feature in your application, set it to `False` to cause all client connections to close immediately after a response is sent, regardless of the Keep-Alive header on the request.

The amount of time the server holds the TCP connection open is decided by the server itself. In Sanic, that value is configured using the `KEEP_ALIVE_TIMEOUT` value. By default, it is set to 5 seconds, this is the same default setting as the Apache HTTP server and is a good balance between allowing enough time for the client to send a new request, and not holding open too many connections at once. Do not exceed 75 seconds unless you know your clients are using a browser which supports TCP connections held open for that long.

For reference:

```
Apache httpd server default keepalive timeout = 5 seconds
Nginx server default keepalive timeout = 75 seconds
Nginx performance tuning guidelines uses keepalive = 15 seconds
IE (5-9) client hard keepalive limit = 60 seconds
Firefox client hard keepalive limit = 115 seconds
Opera 11 client hard keepalive limit = 120 seconds
Chrome 13+ client keepalive limit > 300+ seconds
```

CHAPTER 11

Cookies

Cookies are pieces of data which persist inside a user's browser. Sanic can both read and write cookies, which are stored as key-value pairs.

: Cookies can be freely altered by the client. Therefore you cannot just store data such as login information in cookies as-is, as they can be freely altered by the client. To ensure data you store in cookies is not forged or tampered with by the client, use something like itsdangerous to cryptographically sign the data.

11.1 Reading cookies

A user's cookies can be accessed via the `Request` object's `cookies` dictionary.

```
from sanic.response import text

@app.route("/cookie")
async def test(request):
    test_cookie = request.cookies.get('test')
    return text("Test cookie set to: {}".format(test_cookie))
```

11.2 Writing cookies

When returning a response, cookies can be set on the `Response` object.

```
from sanic.response import text

@app.route("/cookie")
async def test(request):
    response = text("There's a cookie up in this response")
```

(continues on next page)

()

```
    response.cookies['test'] = 'It worked!'
    response.cookies['test']['domain'] = '.gotta-go-fast.com'
    response.cookies['test']['httponly'] = True
    return response
```

11.3 Deleting cookies

Cookies can be removed semantically or explicitly.

```
from sanic.response import text

@app.route("/cookie")
async def test(request):
    response = text("Time to eat some cookies muahaha")

    # This cookie will be set to expire in 0 seconds
    del response.cookies['kill_me']

    # This cookie will self destruct in 5 seconds
    response.cookies['short_life'] = 'Glad to be here'
    response.cookies['short_life']['max-age'] = 5
    del response.cookies['favorite_color']

    # This cookie will remain unchanged
    response.cookies['favorite_color'] = 'blue'
    response.cookies['favorite_color'] = 'pink'
    del response.cookies['favorite_color']

    return response
```

Response cookies can be set like dictionary values and have the following parameters available:

- `expires` (datetime): The time for the cookie to expire on the client's browser.
- `path` (string): The subset of URLs to which this cookie applies. Defaults to /.
- `comment` (string): A comment (metadata).
- `domain` (string): Specifies the domain for which the cookie is valid. An explicitly specified domain must always start with a dot.
- `max-age` (number): Number of seconds the cookie should live for.
- `secure` (boolean): Specifies whether the cookie will only be sent via HTTPS.
- `httponly` (boolean): Specifies whether the cookie cannot be read by Javascript.

CHAPTER 12

Handler Decorators

Since Sanic handlers are simple Python functions, you can apply decorators to them in a similar manner to Flask. A typical use case is when you want some code to run before a handler's code is executed.

12.1 Authorization Decorator

Let's say you want to check that a user is authorized to access a particular endpoint. You can create a decorator that wraps a handler function, checks a request if the client is authorized to access a resource, and sends the appropriate response.

```
from functools import wraps
from sanic.response import json

def authorized():
    def decorator(f):
        @wraps(f)
        async def decorated_function(request, *args, **kwargs):
            # run some method that checks the request
            # for the client's authorization status
            is_authorized = check_request_for_authorization_status(request)

            if is_authorized:
                # the user is authorized.
                # run the handler method and return the response
                response = await f(request, *args, **kwargs)
                return response
            else:
                # the user is not authorized.
                return json({'status': 'not_authorized'}, 403)
        return decorated_function
    return decorator
```

(continues on next page)

()

```
@app.route("/")
@authorized()
async def test(request):
    return json({status: 'authorized'})
```

CHAPTER 13

Streaming

13.1 Request Streaming

Sanic allows you to get request data by stream, as below. When the request ends, `request.stream.get()` returns `None`. Only post, put and patch decorator have stream argument.

```
from sanic import Sanic
from sanic.views import CompositionView
from sanic.views import HTTPMethodView
from sanic.views import stream as stream_decorator
from sanic.blueprints import Blueprint
from sanic.response import stream, text

bp = Blueprint('blueprint_request_stream')
app = Sanic('request_stream')


class SimpleView(HTTPMethodView):

    @stream_decorator
    async def post(self, request):
        result = ''
        while True:
            body = await request.stream.get()
            if body is None:
                break
            result += body.decode('utf-8')
        return text(result)


@app.post('/stream', stream=True)
async def handler(request):
    async def streaming(response):
        while True:
```

(continues on next page)

()

```
            body = await request.stream.get()
            if body is None:
                break
            body = body.decode('utf-8').replace('1', 'A')
            response.write(body)
    return stream(streaming)


@bp.put('/bp_stream', stream=True)
async def bp_handler(request):
    result = ''
    while True:
        body = await request.stream.get()
        if body is None:
            break
        result += body.decode('utf-8').replace('1', 'A')
    return text(result)


async def post_handler(request):
    result = ''
    while True:
        body = await request.stream.get()
        if body is None:
            break
        result += body.decode('utf-8')
    return text(result)

app.blueprint(bp)
app.add_route(SimpleView.as_view(), '/method_view')
view = CompositionView()
view.add(['POST'], post_handler, stream=True)
app.add_route(view, '/composition_view')


if __name__ == '__main__':
    app.run(host='127.0.0.1', port=8000)
```

13.2 Response Streaming

Sanic allows you to stream content to the client with the `stream` method. This method accepts a coroutine callback which is passed a `StreamingHTTPResponse` object that is written to. A simple example is like follows:

```
from sanic import Sanic
from sanic.response import stream

app = Sanic(__name__)

@app.route("/")
async def test(request):
    async def sample_streaming_fn(response):
        response.write('foo,')
        response.write('bar')
```

(continues on next page)

()

```
    return stream(sample_streaming_fn, content_type='text/csv')
```

This is useful in situations where you want to stream content to the client that originates in an external service, like a database. For example, you can stream database records to the client with the asynchronous cursor that `asyncpg` provides:

```
@app.route("/")
async def index(request):
    async def stream_from_db(response):
        conn = await asyncpg.connect(database='test')
        async with conn.transaction():
            async for record in conn.cursor('SELECT generate_series(0, 10)'):
                response.write(record[0])

    return stream(stream_from_db)
```

CHAPTER 14

Class-Based Views

Class-based views are simply classes which implement response behaviour to requests. They provide a way to compartmentalise handling of different HTTP request types at the same endpoint. Rather than defining and decorating three different handler functions, one for each of an endpoint's supported request type, the endpoint can be assigned a class-based view.

14.1 Defining views

A class-based view should subclass `HTTPMethodView`. You can then implement class methods for every HTTP request type you want to support. If a request is received that has no defined method, a `405: Method not allowed` response will be generated.

To register a class-based view on an endpoint, the `app.add_route` method is used. The first argument should be the defined class with the method `as_view` invoked, and the second should be the URL endpoint.

The available methods are `get`, `post`, `put`, `patch`, and `delete`. A class using all these methods would look like the following.

```
from sanic import Sanic
from sanic.views import HTTPMethodView
from sanic.response import text

app = Sanic('some_name')

class SimpleView(HTTPMethodView):

  def get(self, request):
      return text('I am get method')

  def post(self, request):
      return text('I am post method')

  def put(self, request):
```

(continues on next page)

()

```
        return text('I am put method')

    def patch(self, request):
        return text('I am patch method')

    def delete(self, request):
        return text('I am delete method')

app.add_route(SimpleView.as_view(), '/')
```

You can also use `async` syntax.

```
from sanic import Sanic
from sanic.views import HTTPMethodView
from sanic.response import text

app = Sanic('some_name')

class SimpleAsyncView(HTTPMethodView):

    async def get(self, request):
        return text('I am async get method')

app.add_route(SimpleAsyncView.as_view(), '/')
```

14.2 URL parameters

If you need any URL parameters, as discussed in the routing guide, include them in the method definition.

```
class NameView(HTTPMethodView):

    def get(self, request, name):
        return text('Hello {}'.format(name))

app.add_route(NameView.as_view(), '/<name>')
```

14.3 Decorators

If you want to add any decorators to the class, you can set the `decorators` class variable. These will be applied to the class when `as_view` is called.

```
class ViewWithDecorator(HTTPMethodView):
    decorators = [some_decorator_here]

    def get(self, request, name):
        return text('Hello I have a decorator')

    def post(self, request, name):
        return text("Hello I also have a decorator")
```

(continues on next page)

()

```
app.add_route(ViewWithDecorator.as_view(), '/url')
```

But if you just want to decorate some functions and not all functions, you can do as follows:

```
class ViewWithSomeDecorator(HTTPMethodView):

    @staticmethod
    @some_decorator_here
    def get(request, name):
        return text("Hello I have a decorator")

    def post(self, request, name):
        return text("Hello I don't have any decorators")
```

14.4 URL Building

If you wish to build a URL for an HTTPMethodView, remember that the class name will be the endpoint that you will pass into `url_for`. For example:

```
@app.route('/')
def index(request):
    url = app.url_for('SpecialClassView')
    return redirect(url)


class SpecialClassView(HTTPMethodView):
    def get(self, request):
        return text('Hello from the Special Class View!')


app.add_route(SpecialClassView.as_view(), '/special_class_view')
```

14.5 Using CompositionView

As an alternative to the `HTTPMethodView`, you can use `CompositionView` to move handler functions outside of the view class.

Handler functions for each supported HTTP method are defined elsewhere in the source, and then added to the view using the `CompositionView.add` method. The first parameter is a list of HTTP methods to handle (e.g. `['GET', 'POST']`), and the second is the handler function. The following example shows `CompositionView` usage with both an external handler function and an inline lambda:

```
from sanic import Sanic
from sanic.views import CompositionView
from sanic.response import text

app = Sanic(__name__)

def get_handler(request):
    return text('I am a get method')
```

(continues on next page)

()

```
view = CompositionView()
view.add(['GET'], get_handler)
view.add(['POST', 'PUT'], lambda request: text('I am a post/put method'))

# Use the new view to handle requests to the base URL
app.add_route(view, '/')
```

Note: currently you cannot build a URL for a CompositionView using `url_for`.

CHAPTER 15

Custom Protocols

Note: this is advanced usage, and most readers will not need such functionality.

You can change the behavior of Sanic's protocol by specifying a custom protocol, which should be a subclass of asyncio.protocol. This protocol can then be passed as the keyword argument `protocol` to the `sanic.run` method.

The constructor of the custom protocol class receives the following keyword arguments from Sanic.

- `loop`: an `asyncio`-compatible event loop.
- `connections`: a `set` to store protocol objects. When Sanic receives `SIGINT` or `SIGTERM`, it executes `protocol.close_if_idle` for all protocol objects stored in this set.
- `signal`: a `sanic.server.Signal` object with the `stopped` attribute. When Sanic receives `SIGINT` or `SIGTERM`, `signal.stopped` is assigned `True`.
- `request_handler`: a coroutine that takes a `sanic.request.Request` object and a `response` callback as arguments.
- `error_handler`: a `sanic.exceptions.Handler` which is called when exceptions are raised.
- `request_timeout`: the number of seconds before a request times out.
- `request_max_size`: an integer specifying the maximum size of a request, in bytes.

15.1 Example

An error occurs in the default protocol if a handler function does not return an `HTTPResponse` object.

By overriding the `write_response` protocol method, if a handler returns a string it will be converted to an `HTTPResponse object`.

```
from sanic import Sanic
from sanic.server import HttpProtocol
from sanic.response import text
```

(continues on next page)

()

```
app = Sanic(__name__)


class CustomHttpProtocol(HttpProtocol):

    def __init__(self, *, loop, request_handler, error_handler,
                 signal, connections, request_timeout, request_max_size):
        super().__init__(
            loop=loop, request_handler=request_handler,
            error_handler=error_handler, signal=signal,
            connections=connections, request_timeout=request_timeout,
            request_max_size=request_max_size)

    def write_response(self, response):
        if isinstance(response, str):
            response = text(response)
        self.transport.write(
            response.output(self.request.version)
        )
        self.transport.close()


@app.route('/')
async def string(request):
    return 'string'


@app.route('/1')
async def response(request):
    return text('response')

app.run(host='0.0.0.0', port=8000, protocol=CustomHttpProtocol)
```

CHAPTER 16

SSL Example

Optionally pass in an SSLContext:

```
import ssl
context = ssl.create_default_context(purpose=ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain("/path/to/cert", keyfile="/path/to/keyfile")

app.run(host="0.0.0.0", port=8443, ssl=context)
```

You can also pass in the locations of a certificate and key as a dictionary:

```
ssl = {'cert': "/path/to/cert", 'key': "/path/to/keyfile"}
app.run(host="0.0.0.0", port=8443, ssl=ssl)
```

CHAPTER 17

Logging

Sanic allows you to do different types of logging (access log, error log) on the requests based on the python3 logging API. You should have some basic knowledge on python3 logging if you want to create a new configuration.

17.1 Quick Start

A simple example using default settings would be like this:

```
from sanic import Sanic

app = Sanic('test')

@app.route('/')
async def test(request):
    return response.text('Hello World!')

if __name__ == "__main__":
  app.run(debug=True, access_log=True)
```

To use your own logging config, simply use `logging.config.dictConfig`, or pass `log_config` when you initialize `Sanic` app:

```
app = Sanic('test', log_config=LOGGING_CONFIG)
```

And to close logging, simply assign access_log=False:

```
if __name__ == "__main__":
  app.run(access_log=False)
```

This would skip calling logging functions when handling requests. And you could even do further in production to gain extra speed:

```
if __name__ == "__main__":
  # disable debug messages
  app.run(debug=False, access_log=False)
```

17.2 Configuration

By default, log_config parameter is set to use sanic.log.LOGGING_CONFIG_DEFAULTS dictionary for configuration.

There are three `loggers` used in sanic, and **must be defined if you want to create your own logging configuration**:

- root: Used to log internal messages.
- sanic.error: Used to log error logs.
- sanic.access: Used to log access logs.

17.2.1 Log format:

In addition to default parameters provided by python (asctime, levelname, message), Sanic provides additional parameters for access logger with:

- host (str) request.ip
- request (str) request.method + " " + request.url
- status (int) response.status
- byte (int) len(response.body)

The default access log format is

```
%(asctime)s - (%(name)s)[%(levelname)s][%(host)s]: %(request)s %(message)s %(status)d
↪%(byte)d
```

CHAPTER 18

Testing

Sanic endpoints can be tested locally using the `test_client` object, which depends on the additional aiohttp library.

The `test_client` exposes `get`, `post`, `put`, `delete`, `patch`, `head` and `options` methods for you to run against your application. A simple example (using pytest) is like follows:

```
# Import the Sanic app, usually created with Sanic(__name__)
from external_server import app

def test_index_returns_200():
    request, response = app.test_client.get('/')
    assert response.status == 200

def test_index_put_not_allowed():
    request, response = app.test_client.put('/')
    assert response.status == 405
```

Internally, each time you call one of the `test_client` methods, the Sanic app is run at `127.0.0.1:42101` and your test request is executed against your application, using `aiohttp`.

The `test_client` methods accept the following arguments and keyword arguments:

- `uri` *(default `'/'`)* A string representing the URI to test.
- `gather_request` *(default `True`)* A boolean which determines whether the original request will be returned by the function. If set to `True`, the return value is a tuple of `(request, response)`, if `False` only the response is returned.
- `server_kwargs` *(default `{}`) a dict of additional arguments to pass into `app.run` before the test request is run.
- `debug` *(default `False`)* A boolean which determines whether to run the server in debug mode.

The function further takes the `*request_args` and `**request_kwargs`, which are passed directly to the aiohttp ClientSession request.

For example, to supply data to a GET request, you would do the following:

```
def test_get_request_includes_data():
    params = {'key1': 'value1', 'key2': 'value2'}
    request, response = app.test_client.get('/', params=params)
    assert request.args.get('key1') == 'value1'
```

And to supply data to a JSON POST request:

```
def test_post_json_request_includes_data():
    data = {'key1': 'value1', 'key2': 'value2'}
    request, response = app.test_client.post('/', data=json.dumps(data))
    assert request.json.get('key1') == 'value1'
```

More information about the available arguments to aiohttp can be found in the documentation for ClientSession.

18.1 pytest-sanic

pytest-sanic is a pytest plugin, it helps you to test your code asynchronously. 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"
```

pytest-sanic also provides some useful fixtures, like loop, unused_port, test_server, test_client.

```
@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})

    yield app


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


#########
# Tests #
```

(continues on next page)

()

```
#########

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

CHAPTER 19

Deploying

Deploying Sanic is made simple by the inbuilt webserver. After defining an instance of `sanic.Sanic`, we can call the `run` method with the following keyword arguments:

- `host` *(default `"127.0.0.1"`)*: Address to host the server on.
- `port` *(default `8000`)*: Port to host the server on.
- `debug` *(default `False`)*: Enables debug output (slows server).
- `ssl` *(default `None`)*: `SSLContext` for SSL encryption of worker(s).
- `sock` *(default `None`)*: Socket for the server to accept connections from.
- `workers` *(default `1`)*: Number of worker processes to spawn.
- `loop` *(default `None`)*: An `asyncio`-compatible event loop. If none is specified, Sanic creates its own event loop.
- `protocol` *(default `HttpProtocol`)*: Subclass of asyncio.protocol.

19.1 Workers

By default, Sanic listens in the main process using only one CPU core. To crank up the juice, just specify the number of workers in the `run` arguments.

```
app.run(host='0.0.0.0', port=1337, workers=4)
```

Sanic will automatically spin up multiple processes and route traffic between them. We recommend as many workers as you have available cores.

19.2 Running via command

If you like using command line arguments, you can launch a Sanic server by executing the module. For example, if you initialized Sanic as `app` in a file named `server.py`, you could run the server like so:

```
python -m sanic server.app --host=0.0.0.0 --port=1337 --workers=4
```

With this way of running sanic, it is not necessary to invoke `app.run` in your Python file. If you do, make sure you wrap it so that it only executes when directly run by the interpreter.

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=1337, workers=4)
```

19.3 Running via Gunicorn

Gunicorn 'Green Unicorn' is a WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project.

In order to run Sanic application with Gunicorn, you need to use the special `sanic.worker.GunicornWorker` for Gunicorn `worker-class` argument:

```
gunicorn myapp:app --bind 0.0.0.0:1337 --worker-class sanic.worker.GunicornWorker
```

If your application suffers from memory leaks, you can configure Gunicorn to gracefully restart a worker after it has processed a given number of requests. This can be a convenient way to help limit the effects of the memory leak.

See the Gunicorn Docs for more information.

19.4 Asynchronous support

This is suitable if you *need* to share the sanic process with other applications, in particular the `loop`. However be advised that this method does not support using multiple processes, and is not the preferred way to run the app in general.

Here is an incomplete example (please see `run_async.py` in examples for something more practical):

```
server = app.create_server(host="0.0.0.0", port=8000)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(server)
loop.run_forever()
```

CHAPTER 20

Extensions

A list of Sanic extensions created by the community.

- Sanic-Plugins-Framework: Library for easily creating and using Sanic plugins.
- Sessions: Support for sessions. Allows using redis, memcache or an in memory store.
- CORS: A port of flask-cors.
- Compress: Allows you to easily gzip Sanic responses. A port of Flask-Compress.
- Jinja2: Support for Jinja2 template.
- Sanic JWT: Authentication, JWT, and permission scoping for Sanic.
- OpenAPI/Swagger: OpenAPI support, plus a Swagger UI.
- Pagination: Simple pagination support.
- Motor: Simple motor wrapper.
- Sanic CRUD: CRUD REST API generation with peewee models.
- UserAgent: Add `user_agent` to request
- Limiter: Rate limiting for sanic.
- Sanic EnvConfig: Pull environment variables into your sanic config.
- Babel: Adds i18n/l10n support to Sanic applications with the help of the `Babel` library
- Dispatch: A dispatcher inspired by `DispatcherMiddleware` in werkzeug. Can act as a Sanic-to-WSGI adapter.
- Sanic-OAuth: OAuth Library for connecting to & creating your own token providers.
- sanic-oauth: OAuth Library with many provider and OAuth1/OAuth2 support.
- Sanic-nginx-docker-example: Simple and easy to use example of Sanic behined nginx using docker-compose.
- sanic-graphql: GraphQL integration with Sanic
- sanic-prometheus: Prometheus metrics for Sanic

- Sanic-RestPlus: A port of Flask-RestPlus for Sanic. Full-featured REST API with SwaggerUI generation.
- sanic-transmute: A Sanic extension that generates APIs from python function and classes, and also generates Swagger UI/documentation automatically.
- pytest-sanic: A pytest plugin for Sanic. It helps you to test your code asynchronously.
- jinja2-sanic: a jinja2 template renderer for Sanic.(Documentation)
- GINO: An asyncio ORM on top of SQLAlchemy core, delivered with a Sanic extension. (Documentation)
- Sanic-Auth: A minimal backend agnostic session-based user authentication mechanism for Sanic.
- Sanic-CookieSession: A client-side only, cookie-based session, similar to the built-in session in Flask.
- Sanic-WTF: Sanic-WTF makes using WTForms with Sanic and CSRF (Cross-Site Request Forgery) protection a little bit easier.

CHAPTER 21

Contributing

Thank you for your interest! Sanic is always looking for contributors. If you don't feel comfortable contributing code, adding docstrings to the source files is very appreciated.

21.1 Installation

To develop on sanic (and mainly to just run the tests) it is highly recommend to install from sources.

So assume you have already cloned the repo and are in the working directory with a virtual environment already set up, then run:

```
python setup.py develop && pip install -r requirements-dev.txt
```

21.2 Running tests

To run the tests for sanic it is recommended to use tox like so:

```
tox
```

See it's that simple!

21.3 Pull requests!

So the pull request approval rules are pretty simple:

1. All pull requests must pass unit tests

- All pull requests must be reviewed and approved by at least one current collaborator on the project
- All pull requests must pass flake8 checks

- If you decide to remove/change anything from any common interface a deprecation message should accompany it.
- If you implement a new feature you should have at least one unit test to accompany it.

21.4 Documentation

Sanic's documentation is built using sphinx. Guides are written in Markdown and can be found in the `docs` folder, while the module reference is automatically generated using `sphinx-apidoc`.

To generate the documentation from scratch:

```
sphinx-apidoc -fo docs/_api/ sanic
sphinx-build -b html docs docs/_build
```

The HTML documentation will be created in the `docs/_build` folder.

21.5 Warning

One of the main goals of Sanic is speed. Code that lowers the performance of Sanic without significant gains in usability, security, or features may not be merged. Please don't let this intimidate you! If you have any concerns about an idea, open an issue for discussion and help.

CHAPTER 22

API Reference

22.1 Submodules

22.2 sanic.app module

22.3 sanic.blueprints module

22.4 sanic.config module

22.5 sanic.constants module

22.6 sanic.cookies module

22.7 sanic.exceptions module

22.8 sanic.handlers module

22.9 sanic.log module

22.10 sanic.request module

22.11 sanic.response module

22.12 sanic.router module

22.13 sanic.server module

22.14 sanic.static module

CHAPTER 23

Indices and tables

- genindex
- modindex
- search