
Sanic Documentation

Release 18.12.0

Sanic contributors

Feb 06, 2019

1 Sanic aspires to be simple	3
2 Guides	5
2.1 Getting Started	5
2.2 Configuration	6
2.3 Logging	9
2.4 Request Data	10
2.5 Response	13
2.6 Cookies	15
2.7 Routing	16
2.8 Blueprints	22
2.9 Static Files	26
2.10 Versioning	28
2.11 Exceptions	28
2.12 Middleware And Listeners	30
2.13 WebSocket	32
2.14 Handler Decorators	33
2.15 Streaming	34
2.16 Class-Based Views	36
2.17 Custom Protocols	39
2.18 Sockets	40
2.19 SSL Example	41
2.20 Debug Mode	41
2.21 Testing	42
2.22 Deploying	44
2.23 Extensions	46
2.24 Examples	48
2.25 Version 18.12	65
2.26 Version 0.8	66
2.27 Version 0.1	68
2.28 Contributing	69
2.29 API Reference	70
2.30 Python 3.7 AsyncIO examples	111
3 Module Documentation	113
Python Module Index	115

Sanic is a Flask-like Python 3.5+ web server that's written to go fast. It's based on the work done by the amazing folks at magicstack, and was inspired by [this article](#).

On top of being Flask-like, Sanic supports async request handlers. This means you can use the new shiny async/await syntax from Python 3.5, making your code non-blocking and speedy.

Sanic is developed [on GitHub](#). Contributions are welcome!

CHAPTER 1

Sanic aspires to be simple

```
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)
```


2.1 Getting Started

Make sure you have both `pip` and at least version 3.5 of Python before starting. Sanic uses the new `async/await` syntax, so earlier versions of python won't work.

2.1.1 1. Install Sanic

```
pip3 install sanic
```

To install sanic without `uvloop` or `ujson` using bash, you can provide either or both of these environmental variables using any truthy string like 'y', 'yes', 't', 'true', 'on', '1' and setting the `SANIC_NO_X` (X = `UVLOOP/UJSON`) to true will stop that features installation.

```
SANIC_NO_UVLOOP=true SANIC_NO_UJSON=true pip3 install sanic
```

2.1.2 2. Create a file called `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)
```

2.1.3 3. Run the server

```
python3 main.py
```

2.1.4 4. Check your browser

Open the address `http://0.0.0.0:8000` in your web browser. You should see the message *Hello world!*.

You now have a working Sanic server!

2.2 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.

2.2.1 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.

2.2.2 Loading Configuration

There are several ways how to load configuration.

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)
```

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.

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'
```

2.2.3 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_BUFFER_QUEUE_SIZE	100	Request streaming buffer queue size
REQUEST_TIMEOUT	60	How long a request can take to arrive (sec)
RESPONSE_TIMEOUT	60	How long a response can take to process (sec)

(continues on next page)

(continued from previous page)

KEEP_ALIVE	True	Disables keep-alive when False	
↔			↔
KEEP_ALIVE_TIMEOUT	5	How long to hold a TCP connection open	
↔ (sec)			↔
GRACEFUL_SHUTDOWN_TIMEOUT	15.0	How long to wait to force close non-idle	
↔ connection (sec)			↔
ACCESS_LOG	True	Disable or enable access log	
↔			↔

The different Timeout variables:

REQUEST_TIMEOUT

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 an HTTP 408 response and sends that to the client. Set this parameter's value higher if your clients routinely pass very large request payloads or upload requests very slowly.

RESPONSE_TIMEOUT

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 an HTTP 503 response and sends that to the client. Set this parameter's value higher if your application is likely to have long-running process that delay the generation of a response.

KEEP_ALIVE_TIMEOUT

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

Keep-Alive is a HTTP feature introduced in HTTP 1.1. When sending a HTTP request, the client (usually a web browser application) can set a Keep-Alive header to indicate 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
```

(continues on next page)

(continued from previous page)

```
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
```

2.3 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.

2.3.1 Quick Start

A simple example using default settings would be like this:

```
from sanic import Sanic
from sanic.log import logger
from sanic.response import text

app = Sanic('test')

@app.route('/')
async def test(request):
    logger.info('Here is your log')
    return text('Hello World!')

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

After the server is running, you can see some messages looks like:

```
[2018-11-06 21:16:53 +0800] [24622] [INFO] Goin' Fast @ http://127.0.0.1:8000
[2018-11-06 21:16:53 +0800] [24667] [INFO] Starting worker [24667]
```

You can send a request to server and it will print the log messages:

```
[2018-11-06 21:18:53 +0800] [25685] [INFO] Here is your log
[2018-11-06 21:18:53 +0800] - (sanic.access) [INFO][127.0.0.1:57038]: GET http://
↪localhost:8000/ 200 12
```

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)
```

2.3.2 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**:

Logger Name	Usecase
<code>sanic.root</code>	Used to log internal messages.
<code>sanic.error</code>	Used to log error logs.
<code>sanic.access</code>	Used to log access logs.

Log format:

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

Log Context Parameter	Parameter Value	Datatype
<code>host</code>	<code>request.ip</code>	<code>str</code>
<code>request</code>	<code>request.method + " " + request.url</code>	<code>str</code>
<code>status</code>	<code>response.status</code>	<code>int</code>
<code>byte</code>	<code>len(response.body)</code>	<code>int</code>

The default access log format is `%(asctime)s - %(name)s [%(levelname)s] [%(host)s] : %(request)s %(message)s %(status)d %(byte)d`

2.4 Request Data

When an endpoint receives a HTTP request, the route function is passed a `Request` object.

The following variables are accessible as properties on `Request` objects:

- `json` (any) - JSON body

```
from sanic.response import json

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

- `args` (dict) - Query string variables. A query string is the section of a URL that resembles `?key1=value1&key2=value2`. If that URL were to be parsed, the `args` dictionary would look like `{'key1': ['value1'], 'key2': ['value2']}`. The `request`'s `query_string` variable holds the unparsed string value.

```

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` (dict) - On many cases you would need to access the url arguments in a less packed dictionary. For same previous URL `?key1=value1&key2=value2`, the `raw_args` dictionary would look like `{'key1': 'value1', 'key2': 'value2'}`.
- `files` (dictionary of File objects) - List of files that have a name, body, and 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,
    }

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

```

- `form` (dict) - Posted form variables.

```

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) - Posted raw body. This property allows retrieval of the request's raw data, regardless of content type.

```

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` (dict) - A case-insensitive dictionary that contains the request headers.
- `method` (str) - HTTP method of the request (ie GET, POST).
- `ip` (str) - IP address of the requester.
- `port` (str) - Port address of the requester.
- `socket` (tuple) - (IP, port) of the requester.
- `app` - a reference to the Sanic application object that is handling this request. This is useful when inside blueprints or other handlers in modules that do not have access to the global `app` object.

```

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: The full URL of the request, ie: `http://localhost:8000/posts/1/?foo=bar`
- scheme: The URL scheme associated with the request: `http` or `https`
- host: The host associated with the request: `localhost:8080`
- path: The path of the request: `/posts/1/`
- query_string: The query string of the request: `foo=bar` or a blank string `''`
- uri_template: Template for matching route handler: `/posts/<id>/`
- token: The value of Authorization header: `Basic YWRtaW46YWRtaW4=`

2.4.1 Accessing values using `get` and `getlist`

The request properties which return a dictionary actually return a subclass of `dict` called `RequestParameters`. The key difference when using this object is the distinction between the `get` and `getlist` methods.

- `get(key, default=None)` operates as normal, except that when the value of the given key is a list, *only the first item is returned*.
- `getlist(key, default=None)` operates as normal, *returning the entire list*.

```

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']

```

2.4.2 Accessing the handler name with the `request.endpoint` attribute

The `request.endpoint` attribute holds the handler's name. For instance, the below route will return "hello".

```

from sanic.response import text
from sanic import Sanic

app = Sanic()

@app.get("/")
def hello(request):
    return text(request.endpoint)

```


Or, with a blueprint it will be include both, separated by a period. For example, the below route would return foo.bar:

```
from sanic import Sanic
from sanic import Blueprint
from sanic.response import text

app = Sanic(__name__)
blueprint = Blueprint('foo')

@blueprint.get('/')
async def bar(request):
    return text(request.endpoint)

app.blueprint(blueprint)

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

2.5 Response

Use functions in `sanic.response` module to create responses.

2.5.1 Plain Text

```
from sanic import response

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

2.5.2 HTML

```
from sanic import response

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

2.5.3 JSON

```
from sanic import response

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

2.5.4 File

```
from sanic import response

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

2.5.5 Streaming

```
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')
```

2.5.6 File Streaming

For large files, a combination of File and Streaming above

```
from sanic import response

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

2.5.7 Redirect

```
from sanic import response

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

2.5.8 Raw

Response without encoding the body

```
from sanic import response

@app.route('/raw')
def handle_request(request):
    return response.raw(b'raw data')
```

2.5.9 Modify headers or status

To modify headers or status code, pass the `headers` or `status` argument to those functions:

```
from sanic import response

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

2.6 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.

Warning: 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.

2.6.1 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))
```

2.6.2 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")
    response.cookies['test'] = 'It worked!'
    response.cookies['test']['domain'] = '.gotta-go-fast.com'
    response.cookies['test']['httponly'] = True
    return response
```

2.6.3 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.

2.7 Routing

Routing allows the user to specify handler functions for different URL endpoints.

A basic route looks like the following, where `app` is an instance of the `Sanic` class:

```
from sanic.response import json

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

When the url `http://server.url/` is accessed (the base url of the server), the final `/` is matched by the router to the handler function, `test`, which then returns a JSON object.

Sanic handler functions must be defined using the `async def` syntax, as they are asynchronous functions.

2.7.1 Request parameters

Sanic comes with a basic router that supports request parameters.

To specify a parameter, surround it with angle quotes like so: `<PARAM>`. Request parameters will be passed to the route handler functions as keyword arguments.

```
from sanic.response import text

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

To specify a type for the parameter, add a `:type` after the parameter name, inside the quotes. If the parameter does not match the specified type, Sanic will throw a `NotFound` exception, resulting in a 404: Page not found error on the URL.

```
from sanic.response import text

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

@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))
```

2.7.2 HTTP request types

By default, a route defined on a URL will be available for only GET requests to that URL. However, the `@app.route` decorator accepts an optional parameter, `methods`, which allows the handler function to work with any of the HTTP methods in the list.

```
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))
```

There is also an optional `host` argument (which can be a list or a string). This restricts a route to the host or hosts provided. If there is also a route with no host, it will be the default.

```
@app.route('/get', methods=['GET'], host='example.com')
async def get_handler(request):
```

(continues on next page)

(continued from previous page)

```

        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))

```

There are also shorthand method decorators:

```

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))

```

2.7.3 The `add_route` method

As we have seen, routes are often specified using the `@app.route` decorator. However, this decorator is really just a wrapper for the `app.add_route` method, which is used as follows:

```

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'])

```

2.7.4 URL building with `url_for`

Sanic provides a `url_for` method, to generate URLs based on the handler method name. This is useful if you want to avoid hardcoding url paths into your app; instead, you can just reference the handler name. For example:

```

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

```

(continues on next page)

(continued from previous page)

```

    return redirect(url)

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

```

Other things to keep in mind when using `url_for`:

- Keyword arguments passed to `url_for` that are not request parameters will be included in the URL's query string. For example:

```

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

```

- Multivalue argument can be passed to `url_for`. For example:

```

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

```

- Also some special arguments (`_anchor`, `_external`, `_scheme`, `_method`, `_server`) passed to `url_for` will have special url building (`_method` is not supported now and will be ignored). For example:

```

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

url = app.url_for('post_handler', post_id=5, arg_one='one', _external=True)
# //server/posts/5?arg_one=one
# _external requires you to pass an argument _server or set SERVER_NAME in app.config_
↳if not 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 at once
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

```

- All valid parameters must be passed to `url_for` to build a URL. If a parameter is not supplied, or if a parameter does not match the specified type, a `URLBuildError` will be raised.

2.7.5 WebSocket routes

Routes for the WebSocket protocol can be defined with the `@app.websocket` decorator:

```

@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)

```

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

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

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

Handlers to a WebSocket route are invoked with 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.

WebSocket support requires the `websockets` package by Aymeric Augustin.

2.7.6 About `strict_slashes`

You can make routes strict to trailing slash or not, it's configurable.

```
# 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')

# 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)
```

2.7.7 User defined route name

A custom route name can be used by passing a name argument while registering the route which will override the default route name generated using the handler's `__name__` attribute.

```
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')
```

(continues on next page)

(continued from previous page)

```

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')
# 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'

```

2.7.8 Build URL for static files

Sanic supports using `url_for` method to build static file urls. In case if the static url is pointing to a directory, filename parameter to the `url_for` can be ignored. q

```

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')

```

(continues on next page)

(continued from previous page)

```
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'
```

2.8 Blueprints

Blueprints are objects that can be used for sub-routing within an application. Instead of adding routes to the application instance, blueprints define similar methods for adding routes, which are then registered with the application in a flexible and pluggable manner.

Blueprints are especially useful for larger applications, where your application logic can be broken down into several groups or areas of responsibility.

2.8.1 My First Blueprint

The following shows a very simple blueprint that registers a handler-function at the root / of your application.

Suppose you save this file as `my_blueprint.py`, which can be imported into your main application later.

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

bp = Blueprint('my_blueprint')

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

2.8.2 Registering blueprints

Blueprints must be registered with the application.

```
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)
```

This will add the blueprint to the application and register any routes defined by that blueprint. In this example, the registered routes in the `app.router` will look like:

```
[Route(handler=<function bp_root at 0x7f908382f9d8>, methods=frozenset({'GET'}),
↳pattern=re.compile('^/$'), parameters=[], name='my_blueprint.bp_root', uri='/')]
```

2.8.3 Blueprint groups and nesting

Blueprints may also be registered as part of a list or tuple, where the registrar will recursively cycle through any sub-sequences of blueprints and register them accordingly. The `Blueprint.group` method is provided to simplify this process, allowing a ‘mock’ backend directory structure mimicking what’s seen from the front end. Consider this (quite contrived) example:

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

Initialization of this app’s blueprint hierarchy could go as follows:

```
# 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(static, 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')
```

And registering these blueprints in `app.py` can now be done like so:

```
# app.py
from sanic import Sanic

from .api import api

app = Sanic(__name__)

app.blueprint(api)
```

2.8.4 Using Blueprints

Blueprints have almost the same functionality as an application instance.

WebSocket routes

WebSocket handlers can be registered on a blueprint using the `@bp.websocket` decorator or `bp.add_websocket_route` method.

Middleware

Using blueprints allows you to also register middleware globally.

```
@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')
```

Exceptions

Exceptions can be applied exclusively to blueprints globally.

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

Static files

Static files can be served globally, under the blueprint prefix.

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

bp.static('/web/path', '/folder/to/serve')
```

(continues on next page)

(continued from previous page)

```
# 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'
```

2.8.5 Start and stop

Blueprints can run functions during the start and stop process of the server. If running in multiprocessor mode (more than 1 worker), these are triggered after the workers fork.

Available events are:

- `before_server_start`: Executed before the server begins to accept connections
- `after_server_start`: Executed after the server begins to accept connections
- `before_server_stop`: Executed before the server stops accepting connections
- `after_server_stop`: Executed after the server is stopped and all requests are complete

```
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()
```

2.8.6 Use-case: API versioning

Blueprints can be very useful for API versioning, where one blueprint may point at `/v1/<routes>`, and another pointing at `/v2/<routes>`.

When a blueprint is initialised, it can take an optional `version` argument, which will be prepended to all routes defined on the blueprint. This feature can be used to implement our API versioning scheme.

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

blueprint_v1 = Blueprint('v1', url_prefix='/api', version="v1")
blueprint_v2 = Blueprint('v2', url_prefix='/api', version="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')
```

When we register our blueprints on the app, the routes `/v1/api` and `/v2/api` will now point to the individual blueprints, which allows the creation of *sub-sites* for each API version.

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

app = Sanic(__name__)
app.blueprint(blueprint_v1)
app.blueprint(blueprint_v2)

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

2.8.7 URL Building with `url_for`

If you wish to generate a URL for a route inside of a blueprint, remember that the endpoint name takes the format `<blueprint_name>.<handler_name>`. For example:

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

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

2.9 Static Files

Static files and directories, such as an image file, are served by Sanic when registered with the `app.static()` method. The method takes an endpoint URL and a filename. The file specified will then be accessible via the given endpoint.

```
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')
```

(continues on next page)

(continued from previous page)

```

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

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

```

Note: Sanic does not provide directory index when you serve a static directory.

2.9.1 Virtual Host

The `app.static()` method also support **virtual host**. You can serve your static files with specific **virtual host** with `host` argument. For example:

```

from sanic import Sanic

app = Sanic(__name__)

app.static('/static', './static')
app.static('/example_static', './example_static', host='www.example.com')

```

2.9.2 Streaming Large File

In some cases, you might server large file(ex: videos, images, etc.) with Sanic. You can choose to use **streaming file** rather than download directly.

Here is an example:

```

from sanic import Sanic

app = Sanic(__name__)

app.static('/large_video.mp4', '/home/ubuntu/large_video.mp4', stream_large_
↪files=True)

```

When `stream_large_files` is `True`, Sanic will use `file_stream()` instead of `file()` to serve static files. This will use **1KB** as the default chunk size. And, if needed, you can also use a custom chunk size. For example:

```

from sanic import Sanic

app = Sanic(__name__)

chunk_size = 1024 * 1024 * 8 # Set chunk size to 8KB
app.static('/large_video.mp4', '/home/ubuntu/large_video.mp4', stream_large_
↪files=chunk_size)

```

2.10 Versioning

You can pass the `version` keyword to the route decorators, or to a blueprint initializer. It will result in the `v{version}` url prefix where `{version}` is the version number.

2.10.1 Per route

You can pass a version number to the routes directly.

```
from sanic import response

@app.route('/text', version=1)
def handle_request(request):
    return response.text('Hello world! Version 1')

@app.route('/text', version=2)
def handle_request(request):
    return response.text('Hello world! Version 2')

app.run(port=80)
```

Then with curl:

```
curl localhost/v1/text
curl localhost/v2/text
```

2.10.2 Global blueprint version

You can also pass a version number to the blueprint, which will apply to all routes.

```
from sanic import response
from sanic.blueprints import Blueprint

bp = Blueprint('test', version=1)

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

Then with curl:

```
curl localhost/v1/html
```

2.11 Exceptions

Exceptions can be thrown from within request handlers and will automatically be handled by Sanic. Exceptions take a message as their first argument, and can also take a status code to be passed back in the HTTP response.

2.11.1 Throwing an exception

To throw an exception, simply `raise` the relevant exception from the `sanic.exceptions` module.

```
from sanic.exceptions import ServerError

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

You can also use the `abort` function with the appropriate status code:

```
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")
```

2.11.2 Handling exceptions

To override Sanic's default handling of an exception, the `@app.exception` decorator is used. The decorator expects a list of exceptions to handle as arguments. You can pass `SanicException` to catch them all! The decorated exception handler function must take a `Request` and `Exception` object as arguments.

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

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

You can also add an exception handler as such:

```
from sanic import Sanic

async def server_error_handler(request, exception):
    return text("Oops, server error", status=500)

app = Sanic()
app.error_handler.add(Exception, server_error_handler)
```

In some cases, you might want to add some more error handling functionality to what is provided by default. In that case, you can subclass Sanic's default error handler as such:

```
from sanic import Sanic
from sanic.handlers import ErrorHandler

class CustomErrorHandler(ErrorHandler):
    def default(self, request, exception):
        ''' handles errors that have no error handlers assigned '''
        # You custom error handling logic...
        return super().default(request, exception)
```

(continues on next page)

(continued from previous page)

```
app = Sanic()
app.error_handler = CustomErrorHandler()
```

2.11.3 Useful exceptions

Some of the most useful exceptions are presented below:

- `NotFound`: called when a suitable route for the request isn't found.
- `ServerError`: called when something goes wrong inside the server. This usually occurs if there is an exception raised in user code.

See the `sanic.exceptions` module for the full list of exceptions to throw.

2.12 Middleware And Listeners

Middleware are functions which are executed before or after requests to the server. They can be used to modify the *request to* or *response from* user-defined handler functions.

Additionally, Sanic provides listeners which allow you to run code at various points of your application's lifecycle.

2.12.1 Middleware

There are two types of middleware: request and response. Both are declared using the `@app.middleware` decorator, with the decorator's parameter being a string representing its type: 'request' or 'response'.

- Request middleware receives only the `request` as argument.
- Response middleware receives both the `request` and `response`.

The simplest middleware doesn't modify the request or response at all:

```
@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")
```

2.12.2 Modifying the request or response

Middleware can modify the request or response parameter it is given, *as long as it does not return it*. The following example shows a practical use-case for this.

```
app = Sanic(__name__)

@app.middleware('request')
async def add_key(request):
    # Add a key to request object like dict object
    request['foo'] = 'bar'
```

(continues on next page)

(continued from previous page)

```

@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)

```

The above code will apply the three middleware in order. The first middleware **add_key** will add a new key `foo` into request object. This worked because request object can be manipulated like dict object. Then, the second middleware **custom_banner** will change the HTTP response header `Server` to `Fake-Server`, and the last middleware **prevent_xss** will add the HTTP header for preventing Cross-Site-Scripting (XSS) attacks. These two functions are invoked *after* a user function returns a response.

2.12.3 Responding early

If middleware returns a `HTTPResponse` object, the request will stop processing and the response will be returned. If this occurs to a request before the relevant user route handler is reached, the handler will never be called. Returning a response will also prevent any further middleware from running.

```

@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')

```

2.12.4 Listeners

If you want to execute startup/teardown code as your server starts or closes, you can use the following listeners:

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

These listeners are implemented as decorators on functions which accept the app object as well as the asyncio loop.

For example:

```

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

@app.listener('after_server_start')

```

(continues on next page)

(continued from previous page)

```

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()

```

It's also possible to register a listener using the `register_listener` method. This may be useful if you define your listeners in another module besides the one you instantiate your app in.

```

app = Sanic()

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

app.register_listener(setup_db, 'before_server_start')

```

If you want to schedule a background task to run after the loop has started, Sanic provides the `add_task` method to easily do so.

```

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 will attempt to automatically inject the app, passing it as an argument to the task:

```

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)

```

Or you can pass the app explicitly for the same effect:

```

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))
`

```

2.13 WebSocket

Sanic provides an easy to user abstraction on top of *websockets*. To setup a WebSocket:

```

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

```

(continues on next page)

(continued from previous page)

```

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 is invoked with 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

```

app.config.WEBSOCKET_MAX_SIZE = 2 ** 20
app.config.WEBSOCKET_MAX_QUEUE = 32
app.config.WEBSOCKET_READ_LIMIT = 2 ** 16
app.config.WEBSOCKET_WRITE_LIMIT = 2 ** 16

```

Find more in [Configuration](#) section.

2.14 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.

2.14.1 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

```

(continues on next page)

(continued from previous page)

```

    # 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

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

```

2.15 Streaming

2.15.1 Request Streaming

Sanic allows you to get request data by stream, as below. When the request ends, `await request.stream.read()` 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.read()
            if body is None:
                break
            result += body.decode('utf-8')
        return text(result)

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

```

(continues on next page)

(continued from previous page)

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

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

# You can also use `bp.add_route()` with stream argument
async def bp_post_handler(request):
    result = ''
    while True:
        body = await request.stream.read()
        if body is None:
            break
        result += body.decode('utf-8').replace('1', 'A')
    return text(result)

bp.add_route(bp_post_handler, '/bp_stream', methods=['POST'], stream=True)

async def post_handler(request):
    result = ''
    while True:
        body = await request.stream.read()
        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)
```

2.15.2 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):
        await response.write('foo,')
        await response.write('bar')

    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)'):
                await response.write(record[0])

    return stream(stream_from_db)
```

2.16 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.

2.16.1 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
```

(continues on next page)

(continued from previous page)

```
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):
        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(), '/')
```

2.16.2 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>')
```

2.16.3 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")

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")

```

2.16.4 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')

```

2.16.5 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

```

(continues on next page)

(continued from previous page)

```

app = Sanic(__name__)

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

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`.

2.17 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 call-back 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.

2.17.1 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

app = Sanic(__name__)

```

(continues on next page)

(continued from previous page)

```

class CustomHttpProtocol(Protocol):

    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)

```

2.18 Sockets

Sanic can use the python `socket` module to accommodate non IPv4 sockets.

IPv6 example:

```

from sanic import Sanic
from sanic.response import json
import socket

sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
sock.bind([':::', 7777])

app = Sanic()

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

if __name__ == "__main__":
    app.run(sock=sock)

```

to test IPv6 `curl -g -6 "http://[::1]:7777/"`

UNIX socket example:

```
import signal
import sys
import socket
import os
from sanic import Sanic
from sanic.response import json

server_socket = '/tmp/sanic.sock'

sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
sock.bind(server_socket)

app = Sanic()

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

def signal_handler(sig, frame):
    print('Exiting')
    os.unlink(server_socket)
    sys.exit(0)

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

to test UNIX: `curl -v --unix-socket /tmp/sanic.sock http://localhost/hello`

2.19 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)
```

2.20 Debug Mode

When enabling Sanic's debug mode, Sanic will provide a more verbose logging output and by default will enable the Auto Reload feature.

Warning: Sanic's debug mode will slow down the server's performance and is therefore advised to enable it only in development environments.

2.20.1 Setting the debug mode

By setting the debug mode a more verbose output from Sanic will be outputted and the Automatic Reloader will be activated.

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

app = Sanic()

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

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

2.20.2 Manually setting auto reload

Sanic offers a way to enable or disable the Automatic Reloader manually, the `auto_reload` argument will activate or deactivate the Automatic Reloader.

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

app = Sanic()

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

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

2.21 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
```

(continues on next page)

(continued from previous page)

```
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](#).

2.21.1 pytest-sanic

`pytest-sanic` is a `pytest` plugin, it helps you to test your code asynchronously. Just write tests like,

```
async def test_sanit_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 #
#####

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}

```

2.22 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`.
- `access_log` (*default True*): Enables log on handling requests (significantly slows server).

2.22.1 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.

2.22.2 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)
```

2.22.3 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.

2.22.4 Disable debug logging

To improve the performance add `debug=False` and `access_log=False` in the `run` arguments.

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

Running via Gunicorn you can set Environment variable `SANIC_ACCESS_LOG="False"`

```
env SANIC_ACCESS_LOG="False" gunicorn myapp:app --bind 0.0.0.0:1337 --worker-class_
↳sanic.worker.GunicornWorker --log-level warning
```

Or you can rewrite app config directly

```
app.config.ACCESS_LOG = False
```

2.22.5 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()
```

2.23 Extensions

A list of Sanic extensions created by the community.

2.23.1 Extension and Plugin Development

- **Sanic-Plugins-Framework**: Library for easily creating and using Sanic plugins.
- **sanic-script**: An extension for Sanic that adds support for writing commands to your application.

2.23.2 Security

- **Sanic JWT**: Authentication, JWT, and permission scoping for Sanic.
- **Secure**: `Secure` is a lightweight package that adds optional security headers and cookie attributes for Python web frameworks.
- **Sessions**: Support for sessions. Allows using redis, memcache or an in memory store.
- **CORS**: A port of flask-cors.
- **Sanic-JWT-Extended**: Provides extended JWT support for
- **UserAgent**: Add `user_agent` to request
- **Limiter**: Rate limiting for sanic.
- **Sanic-OAuth**: OAuth Library for connecting to & creating your own token providers.
- **sanic-oauth**: OAuth Library with many provider and OAuth1/OAuth2 support.
- **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.

2.23.3 Documentation

- **OpenAPI/Swagger:** OpenAPI support, plus a Swagger UI.
- **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.

2.23.4 ORM and Database Integration

- **Motor:** Simple motor wrapper.
- **Sanic CRUD:** CRUD REST API generation with peewee models.
- **sanic-graphql:** GraphQL integration with Sanic
- **GINO:** An asyncio ORM on top of SQLAlchemy core, delivered with a Sanic extension. ([Documentation](#))

2.23.5 Unit Testing

- **pytest-sanic:** A pytest plugin for Sanic. It helps you to test your code asynchronously.

2.23.6 Project Creation Template

- **cookiecutter-sanic:** Get your sanic application up and running in a matter of second in a well defined project structure. Batteries included for deployment, unit testing, automated release management and changelog generation.

2.23.7 Templating

- **Sanic-WTF:** Sanic-WTF makes using WTForms with Sanic and CSRF (Cross-Site Request Forgery) protection a little bit easier.
- **Jinja2:** Support for Jinja2 template.
- **jinja2-sanic:** a jinja2 template renderer for Sanic. ([Documentation](#))

2.23.8 API Helper Utilities

- **sanic-sse:** [Server-Sent Events](#) implementation for Sanic.
- **Compress:** Allows you to easily gzip Sanic responses. A port of Flask-Compress.
- **Pagination:** Simple pagination support.
- **Sanic EnvConfig:** Pull environment variables into your sanic config.

2.23.9 i18n/l10n Support

- **Babel:** Adds i18n/l10n support to Sanic applications with the help of the [Babel](#) library

2.23.10 Custom Middlewares

- `Dispatch`: A dispatcher inspired by `DispatcherMiddleware` in `werkzeug`. Can act as a Sanic-to-WSGI adapter.

2.23.11 Monitoring and Reporting

- `sanic-prometheus`: Prometheus metrics for Sanic
- `sanic-zipkin`: Easily report request/function/RPC traces to `zipkin/jaeger`, through `aiozipkin`.

2.23.12 Sample Applications

- `Sanic-nginx-docker-example`: Simple and easy to use example of Sanic behind nginx using `docker-compose`.

2.24 Examples

This section of the documentation is a simple collection of example code that can help you get a quick start on your application development. Most of these examples are categorized and provide you with a link to the working code example in the [Sanic Repository](#)

2.24.1 Basic Examples

This section of the examples are a collection of code that provide a simple use case example of the `sanic` application.

Simple Apps

A simple `sanic` application with a single `async` method with `text` and `json` type response.

```
from sanic import Sanic
from sanic import response as res

app = Sanic(__name__)

@app.route("/")
async def test(req):
    return res.text("I\'m a teapot", status=418)

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

```
from sanic import Sanic
from sanic import response

app = Sanic(__name__)

@app.route("/")
```

(continues on next page)

(continued from previous page)

```
async def test(request):
    return response.json({"test": True})

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

Simple App with Sanic Views

Showcasing the simple mechanism of using `sanic.views.HTTPMethodView` as well as a way to extend the same into providing a custom `async` behavior for view.

```
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):
        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')

class SimpleAsyncView(HTTPMethodView):

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

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

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

app.add_route(SimpleView.as_view(), '/')
app.add_route(SimpleAsyncView.as_view(), '/async')

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

URL Redirect

```
from sanic import Sanic
from sanic import response

app = Sanic(__name__)

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

@app.route('/redirect')
async def test(request):
    return response.json({"Redirected": True})

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

Named URL redirection

Sanic provides an easy to use way of redirecting the requests via a helper method called `url_for` that takes a unique url name as argument and returns you the actual route assigned for it. This will help in simplifying the efforts required in redirecting the user between different section of the application.

```
from sanic import Sanic
from sanic import response

app = Sanic(__name__)

@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 response.redirect(url)

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

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

Blueprints

Sanic provides an amazing feature to group your APIs and routes under a logical collection that can easily be imported and plugged into any of your sanic application and it's called `blueprints`

```

from sanic import Blueprint, Sanic
from sanic.response import file, json

app = Sanic(__name__)
blueprint = Blueprint('name', url_prefix='/my_blueprint')
blueprint2 = Blueprint('name2', url_prefix='/my_blueprint2')
blueprint3 = Blueprint('name3', url_prefix='/my_blueprint3')

@blueprint.route('/foo')
async def foo(request):
    return json({'msg': 'hi from blueprint'})

@blueprint2.route('/foo')
async def foo2(request):
    return json({'msg': 'hi from blueprint2'})

@blueprint3.route('/foo')
async def index(request):
    return await file('websocket.html')

@app.websocket('/feed')
async def foo3(request, ws):
    while True:
        data = 'hello!'
        print('Sending: ' + data)
        await ws.send(data)
        data = await ws.recv()
        print('Received: ' + data)

app.blueprint(blueprint)
app.blueprint(blueprint2)
app.blueprint(blueprint3)

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

```

Logging Enhancements

Even though Sanic comes with a battery of Logging support it allows the end users to customize the way logging is handled in the application runtime.

```

from sanic import Sanic
from sanic import response
import logging

logging_format = "[% (asctime)s] %(process)d-% (levelname)s "
logging_format += "% (module)s::%(funcName)s():l%(lineno)d: "
logging_format += "% (message)s"

logging.basicConfig(
    format=logging_format,
    level=logging.DEBUG
)

```

(continues on next page)

(continued from previous page)

```

log = logging.getLogger()

# Set logger to override default basicConfig
sanic = Sanic()

@sanic.route("/")
def test(request):
    log.info("received request; responding with 'hey'")
    return response.text("hey")

sanic.run(host="0.0.0.0", port=8000)

```

The following sample provides an example code that demonstrates the usage of `sanic.app.Sanic.middleware()` in order to provide a mechanism to assign a unique request ID for each of the incoming requests and log them via `aiotask-context`.

```

'''
Based on example from https://github.com/Skyscanner/aiotask-context
and `examples/{override_logging,run_async}.py`.

Needs https://github.com/Skyscanner/aiotask-context/tree/
↪52efbc21e2e1def2d52abb9a8e951f3ce5e6f690 or newer

$ pip install git+https://github.com/Skyscanner/aiotask-context.git
'''

import asyncio
import uuid
import logging
from signal import signal, SIGINT

from sanic import Sanic
from sanic import response

import uvloop
import aiotask_context as context

log = logging.getLogger(__name__)

class RequestIdFilter(logging.Filter):
    def filter(self, record):
        record.request_id = context.get('X-Request-ID')
        return True

LOG_SETTINGS = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'DEBUG',
            'formatter': 'default',
            'filters': ['requestid'],

```

(continues on next page)

(continued from previous page)

```

    },
  },
  'filters': {
    'requestid': {
      '()': RequestIdFilter,
    },
  },
  'formatters': {
    'default': {
      'format': '%(asctime)s %(levelname)s %(name)s:%(lineno)d %(request_id)s |
→%(message)s',
    },
  },
  'loggers': {
    '': {
      'level': 'DEBUG',
      'handlers': ['console'],
      'propagate': True
    },
  }
}

app = Sanic(__name__, log_config=LOG_SETTINGS)

@app.middleware('request')
async def set_request_id(request):
    request_id = request.headers.get('X-Request-ID') or str(uuid.uuid4())
    context.set("X-Request-ID", request_id)

@app.route("/")
async def test(request):
    log.debug('X-Request-ID: %s', context.get('X-Request-ID'))
    log.info('Hello from test!')
    return response.json({"test": True})

if __name__ == '__main__':
    asyncio.set_event_loop(uvloop.new_event_loop())
    server = app.create_server(host="0.0.0.0", port=8000)
    loop = asyncio.get_event_loop()
    loop.set_task_factory(context.task_factory)
    task = asyncio.ensure_future(server)
    try:
        loop.run_forever()
    except:
        loop.stop()

```

Sanic Streaming Support

Sanic framework comes with in-built support for streaming large files and the following code explains the process to setup a Sanic application with streaming support.

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

(continues on next page)

(continued from previous page)

```

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='0.0.0.0', port=8000)

```

Sample Client app to show the usage of streaming application by a client code.

```

import requests

# Warning: This is a heavy process.

data = ""
for i in range(1, 250000):
    data += str(i)

r = requests.post('http://0.0.0.0:8000/stream', data=data)
print(r.text)

```

Sanic Concurrency Support

Sanic supports the ability to start an app with multiple worker support. However, it's important to be able to limit the concurrency per process/loop in order to ensure an efficient execution. The following section of the code provides a brief example of how to limit the concurrency with the help of `asyncio.Semaphore`.

```

from sanic import Sanic
from sanic.response import json

import asyncio
import aiohttp

app = Sanic(__name__)

sem = None

@app.listener('before_server_start')
def init(sanic, loop):
    global sem
    concurrency_per_worker = 4
    sem = asyncio.Semaphore(concurrency_per_worker, loop=loop)

    async def bounded_fetch(session, url):
        """
        Use session object to perform 'get' request on url
        """
        async with sem, session.get(url) as response:
            return await response.json()

@app.route("/")
async def test(request):

```

(continues on next page)

(continued from previous page)

```

"""
Download and serve example JSON
"""
url = "https://api.github.com/repos/channelcat/sanic"

async with aiohttp.ClientSession() as session:
    response = await bounded_fetch(session, url)
    return json(response)

app.run(host="0.0.0.0", port=8000, workers=2)

```

Sanic Deployment via Docker

Deploying a sanic app via docker and docker-compose is an easy task to achieve and the following example provides a deployment of the sample `simple_server.py`

```

FROM python:3.5
MAINTAINER Channel Cat <channelcat@gmail.com>

ADD . /code
RUN pip3 install git+https://github.com/channelcat/sanic

EXPOSE 8000

WORKDIR /code

CMD ["python", "simple_server.py"]

```

```

version: '2'
services:
  sanic:
    build: .
    ports:
      - "8000:8000"

```

Monitoring and Error Handling

Sanic provides an extendable bare minimum implementation of a global exception handler via `sanic.handlers.ErrorHandler`. This example shows how to extend it to enable some custom behaviors.

```

"""
Example intercepting uncaught exceptions using Sanic's error handler framework.
This may be useful for developers wishing to use Sentry, Airbrake, etc.
or a custom system to log and monitor unexpected errors in production.
First we create our own class inheriting from Handler in sanic.exceptions,
and pass in an instance of it when we create our Sanic instance. Inside this
class' default handler, we can do anything including sending exceptions to
an external service.
"""
from sanic.handlers import ErrorHandler
from sanic.exceptions import SanicException
"""

```

(continues on next page)

(continued from previous page)

```

Imports and code relevant for our CustomHandler class
(Ordinarily this would be in a separate file)
"""

class CustomHandler(ErrorHandler):

    def default(self, request, exception):
        # Here, we have access to the exception object
        # and can do anything with it (log, send to external service, etc)

        # Some exceptions are trivial and built into Sanic (404s, etc)
        if not isinstance(exception, SanicException):
            print(exception)

        # Then, we must finish handling the exception by returning
        # our response to the client
        # For this we can just call the super class' default handler
        return super().default(request, exception)

"""
This is an ordinary Sanic server, with the exception that we set the
server's error_handler to an instance of our CustomHandler
"""

from sanic import Sanic

app = Sanic(__name__)

handler = CustomHandler()
app.error_handler = handler

@app.route("/")
async def test(request):
    # Here, something occurs which causes an unexpected exception
    # This exception will flow to our custom handler.
    raise SanicException('You Broke It!')

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

```

Monitoring using external Service Providers

- LogDNA

```

import logging
import socket
from os import getenv
from platform import node
from uuid import getnode as get_mac

from logdna import LogDNAHandler

```

(continues on next page)

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

log = logging.getLogger('logdna')
log.setLevel(logging.INFO)

def get_my_ip_address(remote_server="google.com"):
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        s.connect((remote_server, 80))
        return s.getsockname()[0]

def get_mac_address():
    h = iter(hex(get_mac())[2:].zfill(12))
    return ":".join(i + next(h) for i in h)

logdna_options = {
    "app": __name__,
    "index_meta": True,
    "hostname": node(),
    "ip": get_my_ip_address(),
    "mac": get_mac_address()
}

logdna_handler = LogDNAHandler(getenv("LOGDNA_API_KEY"), options=logdna_options)

logdna = logging.getLogger(__name__)
logdna.setLevel(logging.INFO)
logdna.addHandler(logdna_handler)

app = Sanic(__name__)

@app.middleware
def log_request(request: Request):
    logdna.info("I was Here with a new Request to URL: {}".format(request.url))

@app.route("/")
def default(request):
    return json({
        "response": "I was here"
    })

if __name__ == "__main__":
    app.run(
        host="0.0.0.0",
        port=getenv("PORT", 8080)
    )
```

- RayGun

```

from os import getenv

from raygun4py.raygunprovider import RaygunSender

from sanic import Sanic
from sanic.exceptions import SanicException
from sanic.handlers import ErrorHandler

class RaygunExceptionReporter(ErrorHandler):

    def __init__(self, raygun_api_key=None):
        super().__init__()
        if raygun_api_key is None:
            raygun_api_key = getenv("RAYGUN_API_KEY")

        self.sender = RaygunSender(raygun_api_key)

    def default(self, request, exception):
        self.sender.send_exception(exception=exception)
        return super().default(request, exception)

raygun_error_reporter = RaygunExceptionReporter()
app = Sanic(__name__, error_handler=raygun_error_reporter)

@app.route("/raise")
async def test(request):
    raise SanicException('You Broke It!')

if __name__ == '__main__':
    app.run(
        host="0.0.0.0",
        port=getenv("PORT", 8080)
    )

```

- Rollbar

```

import rollbar

from sanic.handlers import ErrorHandler
from sanic import Sanic
from sanic.exceptions import SanicException
from os import getenv

rollbar.init(getenv("ROLLBAR_API_KEY"))

class RollbarExceptionHandler(ErrorHandler):

    def default(self, request, exception):
        rollbar.report_message(str(exception))
        return super().default(request, exception)

```

(continues on next page)

(continued from previous page)

```
app = Sanic(__name__, error_handler=RollbarExceptionHandler())

@app.route("/raise")
def create_error(request):
    raise SanicException("I was here and I don't like where I am")

if __name__ == "__main__":
    app.run(
        host="0.0.0.0",
        port=getenv("PORT", 8080)
    )
```

- Sentry

```
from os import getenv

from sentry_sdk import init as sentry_init
from sentry_sdk.integrations.sanic import SanicIntegration

from sanic import Sanic
from sanic.response import json

sentry_init(
    dsn=getenv("SENTRY_DSN"),
    integrations=[SanicIntegration()],
)

app = Sanic(__name__)

# noinspection PyUnusedLocal
@app.route("/working")
async def working_path(request):
    return json({
        "response": "Working API Response"
    })

# noinspection PyUnusedLocal
@app.route("/raise-error")
async def raise_error(request):
    raise Exception("Testing Sentry Integration")

if __name__ == '__main__':
    app.run(
        host="0.0.0.0",
        port=getenv("PORT", 8080)
    )
```

Security

The following sample code shows a simple decorator based authentication and authorization mechanism that can be setup to secure your sanic api endpoints.


```

# -*- coding: utf-8 -*-

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

app = Sanic()

def check_request_for_authorization_status(request):
    # Note: Define your check, for instance cookie, session.
    flag = True
    return flag

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

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

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

```

Sanic Websocket

Sanic provides an ability to easily add a route and map it to a websocket handlers.

```

<!DOCTYPE html>
<html>
  <head>
    <title>WebSocket demo</title>
  </head>
  <body>
    <script>
      var ws = new WebSocket('ws://' + document.domain + ':' + location.port +
↪ '/feed'),

```

(continues on next page)

(continued from previous page)

```

        messages = document.createElement('ul');
        ws.onmessage = function (event) {
            var messages = document.getElementsByTagName('ul')[0],
                message = document.createElement('li'),
                content = document.createTextNode('Received: ' + event.data);
            message.appendChild(content);
            messages.appendChild(message);
        };
        document.body.appendChild(messages);
        window.setInterval(function() {
            data = 'bye!'
            ws.send(data);
            var messages = document.getElementsByTagName('ul')[0],
                message = document.createElement('li'),
                content = document.createTextNode('Sent: ' + data);
            message.appendChild(content);
            messages.appendChild(message);
        }, 1000);
    </script>
</body>
</html>

```

```

from sanic import Sanic
from sanic.response import file

app = Sanic(__name__)

@app.route('/')
async def index(request):
    return await file('websocket.html')

@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, debug=True)

```

vhost Support

```

from sanic import response
from sanic import Sanic
from sanic.blueprints import Blueprint

# Usage

```

(continues on next page)

(continued from previous page)

```

# curl -H "Host: example.com" localhost:8000
# curl -H "Host: sub.example.com" localhost:8000
# curl -H "Host: bp.example.com" localhost:8000/question
# curl -H "Host: bp.example.com" localhost:8000/answer

app = Sanic()
bp = Blueprint("bp", host="bp.example.com")

@app.route('/', host=["example.com",
                    "somethingelse.com",
                    "therestofyourdomains.com"])
async def hello(request):
    return response.text("Some defaults")

@app.route('/', host="sub.example.com")
async def hello(request):
    return response.text("42")

@bp.route("/question")
async def hello(request):
    return response.text("What is the meaning of life?")

@bp.route("/answer")
async def hello(request):
    return response.text("42")

app.blueprint(bp)

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

```

Unit Testing With Parallel Test Run Support

The following example shows you how to get up and running with unit testing sanic application with parallel test execution support provided by the `pytest-xdist` plugin.

```

"""pytest-xdist example for sanic server

Install testing tools:

    $ pip install pytest pytest-xdist

Run with xdist params:

    $ pytest examples/pytest_xdist.py -n 8 # 8 workers
"""
import re
from sanic import Sanic
from sanic.response import text
from sanic.testing import PORT as PORT_BASE, SanicTestClient
import pytest

```

(continues on next page)

(continued from previous page)

```

@pytest.fixture(scope="session")
def test_port(worker_id):
    m = re.search(r'[0-9]+', worker_id)
    if m:
        num_id = m.group(0)
    else:
        num_id = 0
    port = PORT_BASE + int(num_id)
    return port

@pytest.fixture(scope="session")
def app():
    app = Sanic()

    @app.route('/')
    async def index(request):
        return text('OK')

    return app

@pytest.fixture(scope="session")
def client(app, test_port):
    return SanicTestClient(app, test_port)

@pytest.mark.parametrize('run_id', range(100))
def test_index(client, run_id):
    request, response = client._sanic_endpoint_test('get', '/')
    assert response.status == 200
    assert response.text == 'OK'

```

Amending Request Object

The request object in Sanic is a kind of dict object, this means that request object can be manipulated as a regular dict object.

```

from sanic import Sanic
from sanic.response import text
from random import randint

app = Sanic()

@app.middleware('request')
def append_request(request):
    # Add new key with random value
    request['num'] = randint(0, 100)

@app.get('/pop')
def pop_handler(request):

```

(continues on next page)

(continued from previous page)

```
# Pop key from request object
num = request.pop('num')
return text(num)

@app.get('/key_exist')
def key_exist_handler(request):
    # Check the key is exist or not
    if 'num' in request:
        return text('num exist in request')

    return text('num does not exist in request')

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

For more examples and useful samples please visit the [Huge-Sanic's GitHub Page](#)

2.25 Version 18.12

18.12.0

- Changes:
 - Improved codebase test coverage from 81% to 91%.
 - Added `stream_large_files` and `host` examples in `static_file` document
 - Added methods to append and finish body content on `Request` (#1379)
 - Integrated with `.appveyor.yml` for windows ci support
 - Added documentation for `AF_INET6` and `AF_UNIX` socket usage
 - Adopt `black/isort` for codestyle
 - Cancel task when `connection_lost`
 - Simplify request ip and port retrieval logic
 - Handle config error in load config file.
 - Integrate with `codecov` for CI
 - Add missed documentation for config section.
 - Deprecate `Handler.log`
 - Pinned `httptools` requirement to version 0.0.10+
- Fixes:
 - Fix `remove_entity_headers` helper function (#1415)
 - Fix `TypeError` when use `Blueprint.group()` to group blueprint with default `url_prefix`, Use `os.path.normpath` to avoid invalid `url_prefix` like `api//v1 f8a6af1` Rename the `http` module to `helpers` to prevent conflicts with the built-in Python `http` library (fixes #1323)
 - Fix unittests on windows
 - Fix Namespacing of `sanic logger`

- Fix missing quotes in decorator example
- Fix redirect with quoted param
- Fix doc for latest blueprint code
- Fix build of latex documentation relating to markdown lists
- Fix loop exception handling in app.py
- Fix content length mismatch in windows and other platform
- Fix Range header handling for static files (#1402)
- Fix the logger and make it work (#1397)
- Fix type pickle->pickle in multiprocessing test
- Fix pickling blueprints Change the string passed in the “name” section of the namedtuples in Blueprint to match the name of the Blueprint module attribute name. This allows blueprints to be pickled and unpickled, without errors, which is a requirement of running Sanic in multiprocessing mode in Windows. Added a test for pickling and unpickling blueprints Added a test for pickling and unpickling sanic itself Added a test for enabling multiprocessing on an app with a blueprint (only useful to catch this bug if the tests are run on Windows).
- Fix document for logging

2.26 Version 0.8

0.8.3

- Changes:
 - Ownership changed to org ‘huge-success’

0.8.0

- Changes:
 - Add Server-Sent Events extension (Innokenty Lebedev)
 - Graceful handling of request_handler_task cancellation (Ashley Sommer)
 - Sanitize URL before redirection (aveao)
 - Add url_bytes to request (johndoe46)
 - py37 support for travisci (yunstanford)
 - Auto reloader support for OSX (garyo)
 - Add UUID route support (Volodymyr Maksymiv)
 - Add pausable response streams (Ashley Sommer)
 - Add weakref to request slots (vopankov)
 - remove ubuntu 12.04 from test fixture due to deprecation (yunstanford)
 - Allow streaming handlers in add_route (kinware)
 - use travis_retry for tox (Raphael Deem)
 - update aiohttp version for test client (yunstanford)
 - add redirect import for clarity (yingshaoxo)

- Update HTTP Entity headers (Arnulfo Solís)
- Add register_listener method (Stephan Fitzpatrick)
- Remove uvloop/ujson dependencies for Windows (abuckenheimer)
- Content-length header on 204/304 responses (Arnulfo Solís)
- Extend WebSocketProtocol arguments and add docs (Bob Olde Hampsink, yunstanford)
- Update development status from pre-alpha to beta (Maksim Anisenkov)
- KeepAlive Timeout log level changed to debug (Arnulfo Solís)
- Pin pytest to 3.3.2 because of pytest-dev/pytest#3170 (Maksim Anisenkov)
- Install Python 3.5 and 3.6 on docker container for tests (Shahin Azad)
- Add support for blueprint groups and nesting (Elias Tarhini)
- Remove uvloop for windows setup (Aleksandr Kurlov)
- Auto Reload (Yaser Amari)
- Documentation updates/fixups (multiple contributors)
- Fixes:
 - Fix: auto_reload in Linux (Ashley Sommer)
 - Fix: broken tests for aiohttp >= 3.3.0 (Ashley Sommer)
 - Fix: disable auto_reload by default on windows (abuckenheimer)
 - Fix (1143): Turn off access log with gunicorn (hqy)
 - Fix (1268): Support status code for file response (Cosmo Borsky)
 - Fix (1266): Add content_type flag to Sanic.static (Cosmo Borsky)
 - Fix: subprotocols parameter missing from add_websocket_route (ciscorn)
 - Fix (1242): Responses for CI header (yunstanford)
 - Fix (1237): add version constraint for websockets (yunstanford)
 - Fix (1231): memory leak - always release resource (Phillip Xu)
 - Fix (1221): make request truthy if transport exists (Raphael Deem)
 - Fix failing tests for aiohttp>=3.1.0 (Ashley Sommer)
 - Fix try_everything examples (PyManiacGR, kot83)
 - Fix (1158): default to auto_reload in debug mode (Raphael Deem)
 - Fix (1136): ErrorHandler.response handler call too restrictive (Julien Castiaux)
 - Fix: raw requires bytes-like object (cloudship)
 - Fix (1120): passing a list in to a route decorator's host arg (Timothy Ebiuwhe)
 - Fix: Bug in multipart/form-data parser (DirkGuijt)
 - Fix: Exception for missing parameter when value is null (NyanKiyoshi)
 - Fix: Parameter check (Howie Hu)
 - Fix (1089): Routing issue with named parameters and different methods (yunstanford)
 - Fix (1085): Signal handling in multi-worker mode (yunstanford)

- Fix: single quote in readme.rst (Cosven)
- Fix: method typos (Dmitry Dygalo)
- Fix: log_response correct output for ip and port (Wibowo Arindrarto)
- Fix (1042): Exception Handling (Raphael Deem)
- Fix: Chinese URIs (Howie Hu)
- Fix (1079): timeout bug when self.transport is None (Raphael Deem)
- Fix (1074): fix strict_slashes when route has slash (Raphael Deem)
- Fix (1050): add samesite cookie to cookie keys (Raphael Deem)
- Fix (1065): allow add_task after server starts (Raphael Deem)
- Fix (1061): double quotes in unauthorized exception (Raphael Deem)
- Fix (1062): inject the app in add_task method (Raphael Deem)
- Fix: update environment.yml for readthedocs (Eli Uriegas)
- Fix: Cancel request task when response timeout is triggered (Jeong YunWon)
- Fix (1052): Method not allowed response for RFC7231 compliance (Raphael Deem)
- Fix: IPv6 Address and Socket Data Format (Dan Palmer)

Note: Changelog was unmaintained between 0.1 and 0.7

2.27 Version 0.1

- 0.1.7
- Reversed static url and directory arguments to meet spec
- 0.1.6
- Static files
- Lazy Cookie Loading
- 0.1.5
- Cookies
- Blueprint listeners and ordering
- Faster Router
- Fix: Incomplete file reads on medium+ sized post requests
- Breaking: after_start and before_stop now pass sanic as their first argument
- 0.1.4
- Multiprocessing
- 0.1.3
- Blueprint support
- Faster Response processing
- 0.1.1 - 0.1.2

- Struggling to update pypi via CI
- 0.1.0
- Released to public

2.28 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.

2.28.1 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:

```
pip3 install -e '[dev]'
```

2.28.2 Dependency Changes

Sanic doesn't use `requirements*.txt` files to manage any kind of dependencies related to it in order to simplify the effort required in managing the dependencies. Please make sure you have read and understood the following section of the document that explains the way `sanic` manages dependencies inside the `setup.py` file.

Dependency Type	Usage	Installation
requirements	Bare minimum dependencies required for sanic to function	<code>pip3 install -e .</code>
tests_require / extras_require['test']	Dependencies required to run the Unit Tests for sanic	<code>pip3 install -e '[test]'</code>
extras_require['dev']	Additional Development requirements to add for contributing	<code>pip3 install -e '[dev]'</code>
extras_require['docs']	Dependencies required to enable building and enhancing sanic documentation	<code>pip3 install -e '[docs]'</code>

2.28.3 Running tests

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

```
tox
```

See it's that simple!

2.28.4 Pull requests!

So the pull request approval rules are pretty simple:

- 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.

2.28.5 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.

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.

2.29 API Reference

2.29.1 Submodules

2.29.2 `sanic.app` module

```
class sanic.app.Sanic (name=None, router=None, error_handler=None, load_env=True, request_class=None, strict_slashes=False, log_config=None, configure_logging=True)
```

Bases: `object`

```
add_route (handler, uri, methods=frozenset({'GET'}), host=None, strict_slashes=None, version=None, name=None, stream=False)
```

A helper method to register class instance or functions as a handler to the application url routes.

Parameters

- **handler** – function or class instance
- **uri** – path of the URL
- **methods** – list or tuple of methods allowed, these are overridden if using a `HTTPMethodView`
- **host** –
- **strict_slashes** –
- **version** –
- **name** – user defined route name for `url_for`
- **stream** – boolean specifying if the handler is a stream handler

Returns function or class instance

add_task (*task*)

Schedule a task to run later, after the loop has started. Different from `asyncio.ensure_future` in that it does not also return a future, and the actual `ensure_future` call is delayed until before server start.

Parameters **task** – future, coroutine or awaitable

add_websocket_route (*handler, uri, host=None, strict_slashes=None, subprotocols=None, name=None*)

A helper method to register a function as a websocket route.

Parameters

- **handler** – a callable function or instance of a class that can handle the websocket request
- **host** – Host IP or FQDN details
- **uri** – URL path that will be mapped to the websocket handler
- **strict_slashes** – If the API endpoint needs to terminate with a “/” or not
- **subprotocols** – Subprotocols to be used with websocket handshake
- **name** – A unique name assigned to the URL so that it can be used with `url_for()`

Returns Object decorated by `websocket()`

blueprint (*blueprint, **options*)

Register a blueprint on the application.

Parameters

- **blueprint** – Blueprint object or (list, tuple) thereof
- **options** – option dictionary with blueprint defaults

Returns Nothing

converted_response_type (*response*)

No implementation provided.

coroutine create_server (*host: Optional[str] = None, port: Optional[int] = None, debug: bool = False, ssl: Union[dict, ssl.SSLContext, None] = None, sock: Optional[socket.socket] = None, protocol: Type[asyncio.protocols.Protocol] = None, backlog: int = 100, stop_event: Any = None, access_log: Optional[bool] = None, return_asyncio_server=False, asyncio_server_kwargs=None*) → None

Asynchronous version of `run()`.

This method will take care of the operations necessary to invoke the `before_start` events via `trigger_events()` method invocation before starting the `sanic` app in Async mode.

Note: This does not support multiprocessing and is not the preferred way to run a `Sanic` application.

Parameters

- **host** (*str*) – Address to host on
- **port** (*int*) – Port to host on
- **debug** (*bool*) – Enables debug output (slows server)

- **ssl** – SSLContext, or location of certificate and key for SSL encryption of worker(s)

:type ssl:SSLContext or dict :param sock: Socket for the server to accept connections from :type sock: socket :param protocol: Subclass of asyncio Protocol class :type protocol: type[Protocol] :param backlog: a number of unaccepted connections that the system

will allow before refusing new connections

Parameters

- **stop_event** (*None*) – event to be triggered before stopping the app - deprecated
- **access_log** (*bool*) – Enables writing access logs (slows server)
- **return_asyncio_server** (*bool*) – flag that defines whether there's a need to return asyncio.Server or start it serving right away
- **asyncio_server_kwargs** (*dict*) – key-value arguments for asyncio/uvloop create_server method

Returns Nothing

delete (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **DELETE HTTP** method

Parameters

- **uri** – URL to be tagged to **DELETE** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

enable_websocket (*enable=True*)

Enable or disable the support for websocket.

Websocket is enabled automatically if websocket routes are added to the application.

exception (**exceptions*)

Decorate a function to be registered as a handler for exceptions

Parameters **exceptions** – exceptions

Returns decorated function

get (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **GET HTTP** method

Parameters

- **uri** – URL to be tagged to **GET** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version

- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

coroutine handle_request (*request, write_callback, stream_callback*)

Take a request from the HTTP Server and return a response object to be sent back The HTTP Server only expects a response object, so exception handling must be done here

Parameters

- **request** – HTTP Request object
- **write_callback** – Synchronous response function to be called with the response as the only argument
- **stream_callback** – Coroutine that handles streaming a StreamingHTTPResponse if produced by the handler.

Returns Nothing

head (*uri, host=None, strict_slashes=None, version=None, name=None*)

listener (*event*)

Create a listener from a decorated function.

Parameters **event** – event to listen to

loop

Synonymous with `asyncio.get_event_loop()`.

Only supported when using the `app.run` method.

middleware (*middleware_or_request*)

Decorate and register middleware to be called before a request. Can either be called as `@app.middleware` or `@app.middleware('request')`

Param `middleware_or_request`: Optional parameter to use for identifying which type of middleware is being registered.

options (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **OPTIONS HTTP** method

Parameters

- **uri** – URL to be tagged to **OPTIONS** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

patch (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **DELETE HTTP** method

Parameters

- **uri** – URL to be tagged to **PATCH** method of *HTTP*
- **host** – Host IP or FQDN for the service to use

- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

post (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **POST HTTP** method

Parameters

- **uri** – URL to be tagged to **POST** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

put (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **PUT HTTP** method

Parameters

- **uri** – URL to be tagged to **PUT** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

register_blueprint (**args, **kwargs*)

Proxy method provided for invoking the *blueprint()* method

Note: To be deprecated in 1.0. Use *blueprint()* instead.

Parameters

- **args** – Blueprint object or (list, tuple) thereof
- **kwargs** – option dictionary with blueprint defaults

Returns None

register_listener (*listener, event*)

Register the listener for a given event.

Args: listener: callable i.e. `setup_db(app, loop)` event: when to register listener i.e. `'before_server_start'`

Returns: listener

register_middleware (*middleware, attach_to='request'*)

Register an application level middleware that will be attached to all the API URLs registered under this application.

This method is internally invoked by the *middleware()* decorator provided at the app level.

Parameters

- **middleware** – Callback method to be attached to the middleware
- **attach_to** – The state at which the middleware needs to be invoked in the lifecycle of an *HTTP Request*. **request** - Invoke before the request is processed **response** - Invoke before the response is returned back

Returns decorated method

remove_route (*uri, clean_cache=True, host=None*)

This method provides the app user a mechanism by which an already existing route can be removed from the *Sanic* object

Parameters

- **uri** – URL Path to be removed from the app
- **clean_cache** – Instruct sanic if it needs to clean up the LRU route cache
- **host** – IP address or FQDN specific to the host

Returns None

route (*uri, methods=frozenset({'GET'}), host=None, strict_slashes=None, stream=False, version=None, name=None*)

Decorate a function to be registered as a route

Parameters

- **uri** – path of the URL
- **methods** – list or tuple of methods allowed
- **host** –
- **strict_slashes** –
- **stream** –
- **version** –
- **name** – user defined route name for url_for

Returns decorated function

run (*host: Optional[str] = None, port: Optional[int] = None, debug: bool = False, ssl: Union[dict, ssl.SSLContext, None] = None, sock: Optional[socket.socket] = None, workers: int = 1, protocol: Type[asyncio.protocols.Protocol] = None, backlog: int = 100, stop_event: Any = None, register_sys_signals: bool = True, access_log: Optional[bool] = None, **kwargs*) → None

Run the HTTP Server and listen until keyboard interrupt or term signal. On termination, drain connections before closing.

Parameters

- **host** (*str*) – Address to host on
- **port** (*int*) – Port to host on
- **debug** (*bool*) – Enables debug output (slows server)
- **ssl** – SSLContext, or location of certificate and key for SSL encryption of worker(s)

:type ssl:SSLContext or dict :param sock: Socket for the server to accept connections from :type sock: socket :param workers: Number of processes received before it is respected :type workers: int :param protocol: Subclass of asyncio Protocol class :type protocol: type[Protocol] :param backlog: a number of unaccepted connections that the system

will allow before refusing new connections

Parameters

- **stop_event** (*None*) – event to be triggered before stopping the app - deprecated
- **register_sys_signals** (*bool*) – Register SIG* events
- **access_log** (*bool*) – Enables writing access logs (slows server)

Returns Nothing

static (*uri, file_or_directory, pattern='/?.+', use_modified_since=True, use_content_range=False, stream_large_files=False, name='static', host=None, strict_slashes=None, content_type=None*)

Register a root to serve files from. The input can either be a file or a directory. This method will enable an easy and simple way to setup the `Route` necessary to serve the static files.

Parameters

- **uri** – URL path to be used for serving static content
- **file_or_directory** – Path for the Static file/directory with static files
- **pattern** – Regex Pattern identifying the valid static files
- **use_modified_since** – If true, send file modified time, and return not modified if the browser's matches the server's
- **use_content_range** – If true, process header for range requests and sends the file part that is requested
- **stream_large_files** – If true, use the `StreamingHTTPResponse.file_stream()` handler rather than the `HTTPResponse.file()` handler to send the file. If this is an integer, this represents the threshold size to switch to `StreamingHTTPResponse.file_stream()`
- **name** – user defined name used for url_for
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **content_type** – user defined content type for header

Returns None

stop()

This kills the Sanic

test_client

coroutine trigger_events (*events, loop*)

Trigger events (functions or async) :param events: one or more sync or async functions to execute :param loop: event loop

url_for (*view_name: str, **kwargs*)

Build a URL based on a view name and the values provided.

In order to build a URL, all request parameters must be supplied as keyword arguments, and each parameter must pass the test for the specified parameter type. If these conditions are not met, a `URLBuildError` will be thrown.

Keyword arguments that are not request parameters will be included in the output URL's query string.

Parameters

- **view_name** – string referencing the view name
- ****kwargs** – keys and values that are used to build request parameters and query string arguments.

Returns the built URL

Raises: `URLBuildError`

websocket (*uri, host=None, strict_slashes=None, subprotocols=None, name=None*)

Decorate a function to be registered as a websocket route :param uri: path of the URL :param subprotocols: optional list of str with supported subprotocols :param host: :return: decorated function

2.29.3 sanic.blueprints module

class `sanic.blueprints.Blueprint` (*name, url_prefix=None, host=None, version=None, strict_slashes=False*)

Bases: `object`

add_route (*handler, uri, methods=frozenset({'GET'})*, *host=None, strict_slashes=None, version=None, name=None, stream=False*)

Create a blueprint route from a function.

Parameters

- **handler** – function for handling uri requests. Accepts function, or class instance with a `view_class` method.
- **uri** – endpoint at which the route will be accessible.
- **methods** – list of acceptable HTTP methods.
- **host** – IP Address of FQDN for the sanic server to use.
- **strict_slashes** – Enforce the API urls are requested with a training /
- **version** – Blueprint Version
- **name** – user defined route name for `url_for`
- **stream** – boolean specifying if the handler is a stream handler

Returns function or class instance

add_websocket_route (*handler, uri, host=None, version=None, name=None*)

Create a blueprint websocket route from a function.

Parameters

- **handler** – function for handling uri requests. Accepts function, or class instance with a `view_class` method.
- **uri** – endpoint at which the route will be accessible.
- **host** – IP Address of FQDN for the sanic server to use.
- **version** – Blueprint Version

- **name** – Unique name to identify the Websocket Route

Returns function or class instance

delete (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **DELETE HTTP** method

Parameters

- **uri** – URL to be tagged to **DELETE** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

exception (**args, **kwargs*)

This method enables the process of creating a global exception handler for the current blueprint under question.

Parameters

- **args** – List of Python exceptions to be caught by the handler
- **kwargs** – Additional optional arguments to be passed to the exception handler

:return a decorated method to handle global exceptions for any route registered under this blueprint.

get (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **GET HTTP** method

Parameters

- **uri** – URL to be tagged to **GET** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

static group (**blueprints, url_prefix=""*)

Create a list of blueprints, optionally grouping them under a general URL prefix.

Parameters

- **blueprints** – blueprints to be registered as a group
- **url_prefix** – URL route to be prepended to all sub-prefixes

head (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **HEAD HTTP** method

Parameters

- **uri** – URL to be tagged to **HEAD** method of *HTTP*

- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct `sanic.app.Sanic` to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

listener (*event*)

Create a listener from a decorated function.

Parameters **event** – Event to listen to.

middleware (**args, **kwargs*)

Create a blueprint middleware from a decorated function.

Parameters

- **args** – Positional arguments to be used while invoking the middleware
- **kwargs** – optional keyword args that can be used with the middleware.

options (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **OPTIONS HTTP** method

Parameters

- **uri** – URL to be tagged to **OPTIONS** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct `sanic.app.Sanic` to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

patch (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **PATCH HTTP** method

Parameters

- **uri** – URL to be tagged to **PATCH** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct `sanic.app.Sanic` to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

post (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **POST HTTP** method

Parameters

- **uri** – URL to be tagged to **POST** method of *HTTP*
- **host** – Host IP or FQDN for the service to use

- **strict_slashes** – Instruct `sanic.app.Sanic` to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

put (`uri, host=None, strict_slashes=None, stream=False, version=None, name=None`)
Add an API URL under the **PUT HTTP** method

Parameters

- **uri** – URL to be tagged to **PUT** method of **HTTP**
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct `sanic.app.Sanic` to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

register (`app, options`)
Register the blueprint to the sanic app.

Parameters

- **app** – Instance of `sanic.app.Sanic` class
- **options** – Options to be used while registering the blueprint into the app. `url_prefix` - URL Prefix to override the blueprint prefix

route (`uri, methods=frozenset({'GET'})`, `host=None, strict_slashes=None, stream=False, version=None, name=None`)
Create a blueprint route from a decorated function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **methods** – list of acceptable HTTP methods.
- **host** – IP Address of FQDN for the sanic server to use.
- **strict_slashes** – Enforce the API urls are requested with a training /
- **stream** – If the route should provide a streaming support
- **version** – Blueprint Version
- **name** – Unique name to identify the Route

:return a decorated method that when invoked will return an object of type `FutureRoute`

static (`uri, file_or_directory, *args, **kwargs`)
Create a blueprint static route from a decorated function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **file_or_directory** – Static asset.

websocket (*uri, host=None, strict_slashes=None, version=None, name=None*)

Create a blueprint websocket route from a decorated function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **host** – IP Address of FQDN for the sanic server to use.
- **strict_slashes** – Enforce the API urls are requested with a training /
- **version** – Blueprint Version
- **name** – Unique name to identify the Websocket Route

class `sanic.blueprints.FutureException` (*handler, args, kwargs*)

Bases: tuple

args

Alias for field number 1

handler

Alias for field number 0

kwargs

Alias for field number 2

class `sanic.blueprints.FutureListener` (*handler, uri, methods, host*)

Bases: tuple

handler

Alias for field number 0

host

Alias for field number 3

methods

Alias for field number 2

uri

Alias for field number 1

class `sanic.blueprints.FutureMiddleware` (*middleware, args, kwargs*)

Bases: tuple

args

Alias for field number 1

kwargs

Alias for field number 2

middleware

Alias for field number 0

class `sanic.blueprints.FutureRoute` (*handler, uri, methods, host, strict_slashes, stream, version, name*)

Bases: tuple

handler

Alias for field number 0

host

Alias for field number 3

methods

Alias for field number 2

name
Alias for field number 7

stream
Alias for field number 5

strict_slashes
Alias for field number 4

uri
Alias for field number 1

version
Alias for field number 6

class `sanic.blueprints.FutureStatic` (*uri, file_or_directory, args, kwargs*)
Bases: tuple

args
Alias for field number 2

file_or_directory
Alias for field number 1

kwargs
Alias for field number 3

uri
Alias for field number 0

2.29.4 `sanic.config` module

class `sanic.config.Config` (*defaults=None, load_env=True, keep_alive=None*)
Bases: dict

from_envvar (*variable_name*)
Load a configuration from an environment variable pointing to a configuration file.

Parameters *variable_name* – name of the environment variable

Returns bool. True if able to load config, False otherwise.

from_object (*obj*)
Update the values from the given object. Objects are usually either modules or classes.
Just the uppercase variables in that object are stored in the config. Example usage:

```
from yourapplication import default_config
app.config.from_object(default_config)
```

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with `from_pyfile()` and ideally from a location not within the package because the package might be installed system wide.

Parameters *obj* – an object holding the configuration

from_pyfile (*filename*)
Update the values in the config from a Python file. Only the uppercase variables in that module are stored in the config.

Parameters *filename* – an absolute path to the config file

`load_environment_vars` (*prefix*='SANIC_')

Looks for prefixed environment variables and applies them to the configuration if present.

2.29.5 `sanic.constants` module

2.29.6 `sanic.cookies` module

class `sanic.cookies.Cookie` (*key*, *value*)

Bases: `dict`

A stripped down version of `Morsel` from `SimpleCookie` #gottagofast

encode (*encoding*)

Encode the cookie content in a specific type of encoding instructed by the developer. Leverages the `str.encode()` method provided by python.

This method can be used to encode and embed `utf-8` content into the cookies.

Parameters `encoding` – Encoding to be used with the cookie

Returns Cookie encoded in a codec of choosing.

Except `UnicodeEncodeError`

class `sanic.cookies.CookieJar` (*headers*)

Bases: `dict`

`CookieJar` dynamically writes headers as cookies are added and removed It gets around the limitation of one header per name by using the `MultiHeader` class to provide a unique key that encodes to `Set-Cookie`.

2.29.7 `sanic.exceptions` module

exception `sanic.exceptions.ContentRangeError` (*message*, *content_range*)

Bases: `sanic.exceptions.SanicException`

status_code = 416

exception `sanic.exceptions.FileNotFound` (*message*, *path*, *relative_url*)

Bases: `sanic.exceptions.NotFound`

exception `sanic.exceptions.Forbidden` (*message*, *status_code*=None)

Bases: `sanic.exceptions.SanicException`

status_code = 403

exception `sanic.exceptions.HeaderNotFound` (*message*, *status_code*=None)

Bases: `sanic.exceptions.InvalidUsage`

exception `sanic.exceptions.InvalidRangeType` (*message*, *content_range*)

Bases: `sanic.exceptions.ContentRangeError`

exception `sanic.exceptions.InvalidUsage` (*message*, *status_code*=None)

Bases: `sanic.exceptions.SanicException`

status_code = 400

exception `sanic.exceptions.MethodNotSupported` (*message*, *method*, *allowed_methods*)

Bases: `sanic.exceptions.SanicException`

status_code = 405

exception `sanic.exceptions.NotFound` (*message*, *status_code=None*)

Bases: `sanic.exceptions.SanicException`

status_code = 404

exception `sanic.exceptions.PayloadTooLarge` (*message*, *status_code=None*)

Bases: `sanic.exceptions.SanicException`

status_code = 413

exception `sanic.exceptions.PyFileError` (*file*)

Bases: `Exception`

exception `sanic.exceptions.RequestTimeout` (*message*, *status_code=None*)

Bases: `sanic.exceptions.SanicException`

The Web server (running the Web site) thinks that there has been too long an interval of time between 1) the establishment of an IP connection (socket) between the client and the server and 2) the receipt of any data on that socket, so the server has dropped the connection. The socket connection has actually been lost - the Web server has ‘timed out’ on that particular socket connection.

status_code = 408

exception `sanic.exceptions.SanicException` (*message*, *status_code=None*)

Bases: `Exception`

exception `sanic.exceptions.ServerError` (*message*, *status_code=None*)

Bases: `sanic.exceptions.SanicException`

status_code = 500

exception `sanic.exceptions.ServiceUnavailable` (*message*, *status_code=None*)

Bases: `sanic.exceptions.SanicException`

The server is currently unavailable (because it is overloaded or down for maintenance). Generally, this is a temporary state.

status_code = 503

exception `sanic.exceptions.URLBuildError` (*message*, *status_code=None*)

Bases: `sanic.exceptions.ServerError`

exception `sanic.exceptions.Unauthorized` (*message*, *status_code=None*, *scheme=None*,
***kwargs*)

Bases: `sanic.exceptions.SanicException`

Unauthorized exception (401 HTTP status code).

Parameters

- **message** – Message describing the exception.
- **status_code** – HTTP Status code.
- **scheme** – Name of the authentication scheme to be used.

When present, `kwargs` is used to complete the WWW-Authentication header.

Examples:

```
# With a Basic auth-scheme, realm MUST be present:
raise Unauthorized("Auth required.",
                  scheme="Basic",
                  realm="Restricted Area")
```

(continues on next page)

(continued from previous page)

```

# With a Digest auth-scheme, things are a bit more complicated:
raise Unauthorized("Auth required.",
                  scheme="Digest",
                  realm="Restricted Area",
                  qop="auth, auth-int",
                  algorithm="MD5",
                  nonce="abcdef",
                  opaque="zyxwvu")

# With a Bearer auth-scheme, realm is optional so you can write:
raise Unauthorized("Auth required.", scheme="Bearer")

# or, if you want to specify the realm:
raise Unauthorized("Auth required.",
                  scheme="Bearer",
                  realm="Restricted Area")

```

```
status_code = 401
```

```
sanic.exceptions.abort(status_code, message=None)
```

Raise an exception based on SanicException. Returns the HTTP response message appropriate for the given status code, unless provided.

Parameters

- **status_code** – The HTTP status code to return.
- **message** – The HTTP response body. Defaults to the messages in response.py for the given status code.

```
sanic.exceptions.add_status_code(code)
```

Decorator used for adding exceptions to *SanicException*.

2.29.8 sanic.handlers module

```
class sanic.handlers.ContentRangeHandler(request, stats)
```

Bases: object

A mechanism to parse and process the incoming request headers to extract the content range information.

Parameters

- **request** (*sanic.request.Request*) – Incoming api request
- **stats** (*posix.stat_result*) – Stats related to the content

Variables

- **start** – Content Range start
- **end** – Content Range end
- **size** – Length of the content
- **total** – Total size identified by the *posix.stat_result* instance
- **ContentRangeHandler.headers** – Content range header dict

```
end
```

```
headers
```

size
start
total

class `sanic.handlers.ErrorHandler`

Bases: `object`

Provide `sanic.app.Sanic` application with a mechanism to handle and process any and all uncaught exceptions in a way the application developer will set fit.

This error handling framework is built into the core that can be extended by the developers to perform a wide range of tasks from recording the error stats to reporting them to an external service that can be used for realtime alerting system.

add (*exception, handler*)

Add a new exception handler to an already existing handler object.

Parameters

- **exception** (`sanic.exceptions.SanicException` or `Exception`) – Type of exception that need to be handled
- **handler** (`function`) – Reference to the method that will handle the exception

Returns `None`

cached_handlers = `None`

default (*request, exception*)

Provide a default behavior for the objects of `ErrorHandler`. If a developer chooses to extent the `ErrorHandler` they can provide a custom implementation for this method to behave in a way they see fit.

Parameters

- **request** (`sanic.request.Request`) – Incoming request
- **exception** (`sanic.exceptions.SanicException` or `Exception`) – Exception object

Returns

handlers = `None`

log (*message, level='error'*)

Deprecated, do not use.

lookup (*exception*)

Lookup the existing instance of `ErrorHandler` and fetch the registered handler for a specific type of exception.

This method leverages a dict lookup to speedup the retrieval process.

Parameters **exception** (`sanic.exceptions.SanicException` or `Exception`) – Type of exception

Returns Registered function if found `None` otherwise

response (*request, exception*)

Fetches and executes an exception handler and returns a response object

Parameters

- **request** (`sanic.request.Request`) – Instance of `sanic.request.Request`

- **exception** (*sanic.exceptions.SanicException* or *Exception*) – Exception to handle

Returns Wrap the return value obtained from *default()* or registered handler for that type of exception.

2.29.9 `sanic.log` module

2.29.10 `sanic.request` module

class `sanic.request.File` (*type, body, name*)

Bases: `tuple`

body

Alias for field number 1

name

Alias for field number 2

type

Alias for field number 0

class `sanic.request.Request` (*url_bytes, headers, version, method, transport*)

Bases: `dict`

Properties of an HTTP request such as URL, headers, etc.

app

args

body

body_finish()

body_init()

body_push (*data*)

content_type

cookies

endpoint

files

form

headers

host

ip

json

load_json (*loads=<built-in function loads>*)

match_info

return matched info after resolving route

method

parsed_args

`parsed_files`

`parsed_form`

`parsed_json`

`path`

`port`

`query_string`

`raw_args`

`raw_url`

`remote_addr`

Attempt to return the original client ip based on X-Forwarded-For.

Returns original client ip.

`scheme`

`socket`

`stream`

`token`

Attempt to return the auth header token.

Returns token related to request

`transport`

`uri_template`

`url`

`version`

class `sanic.request.RequestParameters`

Bases: `dict`

Hosts a dict with lists as values where `get` returns the first value of the list and `getlist` returns the whole shebang

get (*name, default=None*)

Return the first value, either the default or actual

getlist (*name, default=None*)

Return the entire list

class `sanic.request.StreamBuffer` (*buffer_size=100*)

Bases: `object`

is_full ()

coroutine put (*payload*)

coroutine read ()

Stop reading when gets None

`sanic.request.parse_multipart_form` (*body, boundary*)

Parse a request body and returns fields and files

Parameters

- **body** – bytes request body
- **boundary** – bytes multipart boundary

Returns fields (RequestParameters), files (RequestParameters)

2.29.11 `sanic.response` module

class `sanic.response.BaseHTTPResponse`

Bases: `object`

cookies

class `sanic.response.HTTPResponse` (*body=None, status=200, headers=None, content_type='text/plain', body_bytes=b''*)

Bases: `sanic.response.BaseHTTPResponse`

body

content_type

cookies

headers

output (*version='1.1', keep_alive=False, keep_alive_timeout=None*)

status

class `sanic.response.StreamingHTTPResponse` (*streaming_fn, status=200, headers=None, content_type='text/plain'*)

Bases: `sanic.response.BaseHTTPResponse`

content_type

get_headers (*version='1.1', keep_alive=False, keep_alive_timeout=None*)

headers

protocol

status

coroutine stream (*version='1.1', keep_alive=False, keep_alive_timeout=None*)

Streams headers, runs the *streaming_fn* callback that writes content to the response body, then finalizes the response body.

streaming_fn

coroutine write (*data*)

Writes a chunk of data to the streaming response.

Parameters *data* – bytes-ish data to be written.

coroutine `sanic.response.file` (*location, status=200, mime_type=None, headers=None, filename=None, _range=None*)

Return a response object with file data.

Parameters

- **location** – Location of file on system.
- **mime_type** – Specific mime_type.
- **headers** – Custom Headers.
- **filename** – Override filename.
- **_range** –

`coroutine` `sanic.response.file_stream` (*location*, *status=200*, *chunk_size=4096*,
mime_type=None, *headers=None*, *filename=None*,
_range=None)

Return a streaming response object with file data.

Parameters

- **location** – Location of file on system.
- **chunk_size** – The size of each chunk in the stream (in bytes)
- **mime_type** – Specific mime_type.
- **headers** – Custom Headers.
- **filename** – Override filename.
- **_range** –

`sanic.response.html` (*body*, *status=200*, *headers=None*)

Returns response object with body in html format.

Parameters

- **body** – Response data to be encoded.
- **status** – Response code.
- **headers** – Custom Headers.

`sanic.response.json` (*body*, *status=200*, *headers=None*, *content_type='application/json'*,
dumps=<built-in function dumps>, ***kwargs*)

Returns response object with body in json format.

Parameters

- **body** – Response data to be serialized.
- **status** – Response code.
- **headers** – Custom Headers.
- **kwargs** – Remaining arguments that are passed to the json encoder.

`sanic.response.raw` (*body*, *status=200*, *headers=None*, *content_type='application/octet-stream'*)

Returns response object without encoding the body.

Parameters

- **body** – Response data.
- **status** – Response code.
- **headers** – Custom Headers.
- **content_type** – the content type (string) of the response.

`sanic.response.redirect` (*to*, *headers=None*, *status=302*, *content_type='text/html; charset=utf-8'*)

Abort execution and cause a 302 redirect (by default).

Parameters

- **to** – path or fully qualified URL to redirect to
- **headers** – optional dict of headers to include in the new request
- **status** – status code (int) of the new request, defaults to 302
- **content_type** – the content type (string) of the response

Returns the redirecting Response

```
sanic.response.stream(streaming_fn, status=200, headers=None, content_type='text/plain; charset=utf-8')
```

Accepts an coroutine *streaming_fn* which can be used to write chunks to a streaming response. Returns a *StreamingHTTPResponse*.

Example usage:

```
@app.route("/")
async def index(request):
    async def streaming_fn(response):
        await response.write('foo')
        await response.write('bar')

    return stream(streaming_fn, content_type='text/plain')
```

Parameters

- **streaming_fn** – A coroutine accepts a response and writes content to that response.
- **mime_type** – Specific mime_type.
- **headers** – Custom Headers.

```
sanic.response.text(body, status=200, headers=None, content_type='text/plain; charset=utf-8')
```

Returns response object with body in text format.

Parameters

- **body** – Response data to be encoded.
- **status** – Response code.
- **headers** – Custom Headers.
- **content_type** – the content type (string) of the response

2.29.12 `sanic.router` module

```
class sanic.router.Parameter (name, cast)
```

Bases: tuple

cast

Alias for field number 1

name

Alias for field number 0

```
exception sanic.router.ParameterNameConflicts
```

Bases: Exception

```
class sanic.router.Route (handler, methods, pattern, parameters, name, uri)
```

Bases: tuple

handler

Alias for field number 0

methods

Alias for field number 1

name
Alias for field number 4

parameters
Alias for field number 3

pattern
Alias for field number 2

uri
Alias for field number 5

exception `sanic.router.RouteDoesNotExist`
Bases: Exception

exception `sanic.router.RouteExists`
Bases: Exception

class `sanic.router.Router`
Bases: object

Router supports basic routing with parameters and method checks

Usage:

```
@sanic.route('/my/url/<my_param>', methods=['GET', 'POST', ...])
def my_route(request, my_param):
    do stuff...
```

or

```
@sanic.route('/my/url/<my_param:my_type>', methods=['GET', 'POST', ...])
def my_route_with_type(request, my_param: my_type):
    do stuff...
```

Parameters will be passed as keyword arguments to the request handling function. Provided parameters can also have a type by appending `:type` to the `<parameter>`. Given parameter must be able to be type-casted to this. If no type is provided, a string is expected. A regular expression can also be passed in as the type. The argument given to the function will always be a string, independent of the type.

add (*uri, methods, handler, host=None, strict_slashes=False, version=None, name=None*)
Add a handler to the route list

Parameters

- **uri** – path to match
- **methods** – sequence of accepted method names. If none are provided, any method is allowed
- **handler** – request handler function. When executed, it should provide a response object.
- **strict_slashes** – strict to trailing slash
- **version** – current version of the route or blueprint. See docs for further details.

Returns Nothing

static check_dynamic_route_exists (*pattern, routes_to_check, parameters*)

Check if a URL pattern exists in a list of routes provided based on the comparison of URL pattern and the parameters.

Parameters

- **pattern** – URL parameter pattern
- **routes_to_check** – list of dynamic routes either hashable or unhashable routes.
- **parameters** – List of *Parameter* items

Returns Tuple of index and route if matching route exists else -1 for index and None for route

find_route_by_view_name

Find a route in the router based on the specified view name.

Parameters

- **view_name** – string of view name to search by
- **kwargs** – additional params, usually for static files

Returns tuple containing (uri, Route)

get (*request*)

Get a request handler based on the URL of the request, or raises an error

Parameters **request** – Request object

Returns handler, arguments, keyword arguments

get_supported_methods (*url*)

Get a list of supported methods for a url and optional host.

Parameters **url** – URL string (including host)

Returns frozenset of supported methods

is_stream_handler (*request*)

Handler for request is stream or not. :param request: Request object :return: bool

parameter_pattern = `re.compile('<(.*?)>')`

classmethod parse_parameter_string (*parameter_string*)

Parse a parameter string into its constituent name, type, and pattern

For example:

```
parse_parameter_string('<param_one:[A-z]>')` ->
('param_one', str, '[A-z]')
```

Parameters **parameter_string** – String to parse

Returns tuple containing (parameter_name, parameter_type, parameter_pattern)

remove (*uri*, *clean_cache=True*, *host=None*)

routes_always_check = None

routes_dynamic = None

routes_static = None

`sanic.router.url_hash` (*url*)

2.29.13 sanic.server module

```
class sanic.server.HttpProtocol (*, loop, request_handler, error_handler, signal=<sanic.server.Signal object>, connections=None, request_timeout=60, response_timeout=60, keep_alive_timeout=5, request_max_size=None, request_buffer_queue_size=100, request_class=None, access_log=True, keep_alive=True, is_request_stream=False, router=None, state=None, debug=False, **kwargs)
```

Bases: `asyncio.protocols.Protocol`

This class provides a basic HTTP implementation of the sanic framework.

access_log

bail_out (*message*, *from_error=False*)

In case if the transport pipes are closed and the sanic app encounters an error while writing data to the transport pipe, we log the error with proper details.

Parameters

- **message** (*str*) – Error message to display
- **from_error** (*bool*) – If the bail out was invoked while handling an exception scenario.

Returns None

coroutine body_append (*body*)

cleanup ()

This is called when KeepAlive feature is used, it resets the connection in order for it to be able to handle receiving another request on the same connection.

close ()

Force close the connection.

close_if_idle ()

Close the connection if a request is not being sent or received

Returns boolean - True if closed, false if staying open

connection_lost (*exc*)

Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

connection_made (*transport*)

Called when a connection is made.

The argument is the transport representing the pipe connection. To receive data, wait for `data_received()` calls. When the connection is closed, `connection_lost()` is called.

connections

data_received (*data*)

Called when some data is received.

The argument is a bytes object.

coroutine drain ()

error_handler

execute_request_handler()

Invoke the request handler defined by the `sanic.app.Sanic.handle_request()` method

Returns None

headers**is_request_stream****keep_alive**

Check if the connection needs to be kept alive based on the params attached to the `_keep_alive` attribute, `Signal.stopped` and `HttpProtocol.parser.should_keep_alive()`

Returns True if connection is to be kept alive False else

keep_alive_timeout**keep_alive_timeout_callback()**

Check if elapsed time since last response exceeds our configured maximum keep alive timeout value and if so, close the transport pipe and let the response writer handle the error.

Returns None

log_response(response)

Helper method provided to enable the logging of responses in case if the `HttpProtocol.access_log` is enabled.

Parameters **response** (`sanic.response.HTTPResponse` or `sanic.response.StreamingHTTPResponse`) – Response generated for the current request

Returns None

loop**on_body(body)****on_header(name, value)****on_headers_complete()****on_message_complete()****on_url(url)****parser****pause_writing()**

Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – `pause_writing()` is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually `resume_writing()` is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, `pause_writing()` is not called – it must go strictly over. Conversely, `resume_writing()` is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through `EventLoop.call_soon()` – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until `pause_writing()` is called).

push_data(data)**request****request_buffer_queue_size**

`request_class`

`request_handler`

`request_max_size`

`request_timeout`

`request_timeout_callback()`

`response_timeout`

`response_timeout_callback()`

`resume_writing()`

Called when the transport's buffer drains below the low-water mark.

See `pause_writing()` for details.

`router`

`signal`

`state`

`coroutine stream_response(response)`

Streams a response to the client asynchronously. Attaches the transport to the response so the response consumer can write to the response as needed.

`transport`

`url`

`write_error(exception)`

`write_response(response)`

Writes response content synchronously to the transport.

class `sanic.server.Signal`

Bases: `object`

`stopped = False`

`sanic.server.serve(host, port, request_handler, error_handler, before_start=None, after_start=None, before_stop=None, after_stop=None, debug=False, request_timeout=60, response_timeout=60, keep_alive_timeout=5, ssl=None, sock=None, request_max_size=None, request_buffer_queue_size=100, reuse_port=False, loop=None, protocol=<class 'sanic.server.HttpProtocol'>, backlog=100, register_sys_signals=True, run_multiple=False, run_async=False, connections=None, signal=<sanic.server.Signal object>, request_class=None, access_log=True, keep_alive=True, is_request_stream=False, router=None, websocket_max_size=None, websocket_max_queue=None, websocket_read_limit=65536, websocket_write_limit=65536, state=None, graceful_shutdown_timeout=15.0, asyncio_server_kwargs=None)`

Start asynchronous HTTP Server on an individual process.

Parameters

- **host** – Address to host on
- **port** – Port to host on
- **request_handler** – Sanic request handler with middleware
- **error_handler** – Sanic error handler with middleware

- **before_start** – function to be executed before the server starts listening. Takes arguments *app* instance and *loop*
- **after_start** – function to be executed after the server starts listening. Takes arguments *app* instance and *loop*
- **before_stop** – function to be executed when a stop signal is received before it is respected. Takes arguments *app* instance and *loop*
- **after_stop** – function to be executed when a stop signal is received after it is respected. Takes arguments *app* instance and *loop*
- **debug** – enables debug output (slows server)
- **request_timeout** – time in seconds
- **response_timeout** – time in seconds
- **keep_alive_timeout** – time in seconds
- **ssl** – SSLContext
- **sock** – Socket for the server to accept connections from
- **request_max_size** – size in bytes, *None* for no limit
- **reuse_port** – *True* for multiple workers
- **loop** – asyncio compatible event loop
- **protocol** – subclass of asyncio protocol class
- **request_class** – Request class to use
- **access_log** – disable/enable access log
- **websocket_max_size** – enforces the maximum size for incoming messages in bytes.
- **websocket_max_queue** – sets the maximum length of the queue that holds incoming messages.
- **websocket_read_limit** – sets the high-water limit of the buffer for incoming bytes, the low-water limit is half the high-water limit.
- **websocket_write_limit** – sets the high-water limit of the buffer for outgoing bytes, the low-water limit is a quarter of the high-water limit.
- **is_request_stream** – disable/enable Request.stream
- **request_buffer_queue_size** – streaming request buffer queue size
- **router** – Router object
- **graceful_shutdown_timeout** – How long take to Force close non-idle connection
- **asyncio_server_kwargs** – key-value args for asyncio/uvloop create_server method

Returns Nothing

`sanic.server.serve_multiple` (*server_settings*, *workers*)

Start multiple server processes simultaneously. Stop on interrupt and terminate signals, and drain connections when complete.

Parameters

- **server_settings** – kw arguments to be passed to the serve function
- **workers** – number of workers to launch

- **stop_event** – if provided, is used as a stop signal

Returns

`sanic.server.trigger_events` (*events, loop*)

Trigger event callbacks (functions or async)

Parameters

- **events** – one or more sync or async functions to execute
- **loop** – event loop

`sanic.server.update_current_time` (*loop*)

Cache the current time, since it is needed at the end of every keep-alive request to update the request timeout time

Parameters *loop* –

Returns

2.29.14 `sanic.static` module

`sanic.static.register` (*app, uri, file_or_directory, pattern, use_modified_since, use_content_range, stream_large_files, name='static', host=None, strict_slashes=None, content_type=None*)

Register a static directory handler with Sanic by adding a route to the router and registering a handler.

Parameters

- **app** – Sanic
- **file_or_directory** – File or directory path to serve from
- **uri** – URL to serve from
- **pattern** – regular expression used to match files in the URL
- **use_modified_since** – If true, send file modified time, and return not modified if the browser's matches the server's
- **use_content_range** – If true, process header for range requests and sends the file part that is requested
- **stream_large_files** – If true, use the `file_stream()` handler rather than the `file()` handler to send the file. If this is an integer, this represents the threshold size to switch to `file_stream()`
- **name** – user defined name used for `url_for`
- **content_type** – user defined content type for header

2.29.15 `sanic.testing` module

`class` `sanic.testing.SanicTestClient` (*app, port=42101*)

Bases: `object`

delete (**args, **kwargs*)

get (**args, **kwargs*)

head (**args, **kwargs*)

```

options (*args, **kwargs)
patch (*args, **kwargs)
post (*args, **kwargs)
put (*args, **kwargs)

```

2.29.16 sanic.views module

class `sanic.views.CompositionView`

Bases: `object`

Simple method-function mapped view for the sanic. You can add handler functions to methods (get, post, put, patch, delete) for every HTTP method you want to support.

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

etc.

If someone tries to use a non-implemented method, there will be a 405 response.

add (*methods, handler, stream=False*)

class `sanic.views.HTTPMethodView`

Bases: `object`

Simple class based implementation of view for the sanic. You should implement methods (get, post, put, patch, delete) for the class to every HTTP method you want to support.

For example:

```

class DummyView(HTTPMethodView):
    def get(self, request, *args, **kwargs):
        return text('I am get method')
    def put(self, request, *args, **kwargs):
        return text('I am put method')

```

etc.

If someone tries to use a non-implemented method, there will be a 405 response.

If you need any url params just mention them in method definition:

```

class DummyView(HTTPMethodView):
    def get(self, request, my_param_here, *args, **kwargs):
        return text('I am get method with %s' % my_param_here)

```

To add the view into the routing you could use

- 1) `app.add_route(DummyView.as_view(), '/')`
- 2) `app.route('/') (DummyView.as_view())`

To add any decorator you could set it into decorators variable

classmethod `as_view` (*class_args, **class_kwargs)

Return view function for use with the routing system, that dispatches request to appropriate handler method.

decorators = []

`dispatch_request` (*request*, **args*, ***kwargs*)

`sanic.views.stream` (*func*)

2.29.17 `sanic.websocket` module

```
class sanic.websocket.WebSocketProtocol (*args, websocket_timeout=10,  
                                         websocket_max_size=None, web-  
                                         socket_max_queue=None, web-  
                                         socket_read_limit=65536, web-  
                                         socket_write_limit=65536, **kwargs)
```

Bases: `sanic.server.HttpProtocol`

`connection_lost` (*exc*)

Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

`data_received` (*data*)

Called when some data is received.

The argument is a bytes object.

`keep_alive_timeout_callback` ()

Check if elapsed time since last response exceeds our configured maximum keep alive timeout value and if so, close the transport pipe and let the response writer handle the error.

Returns None

`request_timeout_callback` ()

`response_timeout_callback` ()

`coroutine websocket_handshake` (*request*, *subprotocols*=None)

`write_response` (*response*)

Writes response content synchronously to the transport.

2.29.18 `sanic.worker` module

2.29.19 Module contents

```
class sanic.Sanic (name=None, router=None, error_handler=None, load_env=True, re-  
                  quest_class=None, strict_slashes=False, log_config=None, config-  
                  ure_logging=True)
```

Bases: `object`

`add_route` (*handler*, *uri*, *methods*=frozenset({'GET'}), *host*=None, *strict_slashes*=None, *ver-*
 sion=None, *name*=None, *stream*=False)

A helper method to register class instance or functions as a handler to the application url routes.

Parameters

- **handler** – function or class instance
- **uri** – path of the URL
- **methods** – list or tuple of methods allowed, these are overridden if using a HTTPMethod-View

- **host** –
- **strict_slashes** –
- **version** –
- **name** – user defined route name for `url_for`
- **stream** – boolean specifying if the handler is a stream handler

Returns function or class instance

add_task (*task*)

Schedule a task to run later, after the loop has started. Different from `asyncio.ensure_future` in that it does not also return a future, and the actual `ensure_future` call is delayed until before server start.

Parameters **task** – future, coroutine or awaitable

add_websocket_route (*handler, uri, host=None, strict_slashes=None, subprotocols=None, name=None*)

A helper method to register a function as a websocket route.

Parameters

- **handler** – a callable function or instance of a class that can handle the websocket request
- **host** – Host IP or FQDN details
- **uri** – URL path that will be mapped to the websocket handler
- **strict_slashes** – If the API endpoint needs to terminate with a “/” or not
- **subprotocols** – Subprotocols to be used with websocket handshake
- **name** – A unique name assigned to the URL so that it can be used with `url_for()`

Returns Object decorated by `websocket()`

blueprint (*blueprint, **options*)

Register a blueprint on the application.

Parameters

- **blueprint** – Blueprint object or (list, tuple) thereof
- **options** – option dictionary with blueprint defaults

Returns Nothing

converted_response_type (*response*)

No implementation provided.

coroutine create_server (*host: Optional[str] = None, port: Optional[int] = None, debug: bool = False, ssl: Union[dict, ssl.SSLContext, None] = None, sock: Optional[socket.socket] = None, protocol: Type[asyncio.protocols.Protocol] = None, backlog: int = 100, stop_event: Any = None, access_log: Optional[bool] = None, return_asyncio_server=False, asyncio_server_kwargs=None*) → None

Asynchronous version of `run()`.

This method will take care of the operations necessary to invoke the `before_start` events via `trigger_events()` method invocation before starting the `sanic` app in Async mode.

Note: This does not support multiprocessing and is not the preferred way to run a `Sanic` application.

Parameters

- **host** (*str*) – Address to host on
- **port** (*int*) – Port to host on
- **debug** (*bool*) – Enables debug output (slows server)
- **ssl** – SSLContext, or location of certificate and key for SSL encryption of worker(s)

:type ssl:SSLContext or dict :param sock: Socket for the server to accept connections from :type sock: socket :param protocol: Subclass of asyncio Protocol class :type protocol: type[Protocol] :param backlog: a number of unaccepted connections that the system

will allow before refusing new connections

Parameters

- **stop_event** (*None*) – event to be triggered before stopping the app - deprecated
- **access_log** (*bool*) – Enables writing access logs (slows server)
- **return_asyncio_server** (*bool*) – flag that defines whether there's a need to return asyncio.Server or start it serving right away
- **asyncio_server_kwargs** (*dict*) – key-value arguments for asyncio/uvloop create_server method

Returns Nothing

delete (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **DELETE HTTP** method

Parameters

- **uri** – URL to be tagged to **DELETE** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

enable_websocket (*enable=True*)

Enable or disable the support for websocket.

Websocket is enabled automatically if websocket routes are added to the application.

exception (**exceptions*)

Decorate a function to be registered as a handler for exceptions

Parameters **exceptions** – exceptions

Returns decorated function

get (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **GET HTTP** method

Parameters

- **uri** – URL to be tagged to **GET** method of *HTTP*

- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

coroutine handle_request (*request*, *write_callback*, *stream_callback*)

Take a request from the HTTP Server and return a response object to be sent back The HTTP Server only expects a response object, so exception handling must be done here

Parameters

- **request** – HTTP Request object
- **write_callback** – Synchronous response function to be called with the response as the only argument
- **stream_callback** – Coroutine that handles streaming a StreamingHTTPResponse if produced by the handler.

Returns Nothing

head (*uri*, *host=None*, *strict_slashes=None*, *version=None*, *name=None*)

listener (*event*)

Create a listener from a decorated function.

Parameters **event** – event to listen to

loop

Synonymous with `asyncio.get_event_loop()`.

Only supported when using the `app.run` method.

middleware (*middleware_or_request*)

Decorate and register middleware to be called before a request. Can either be called as `@app.middleware` or `@app.middleware('request')`

Param `middleware_or_request`: Optional parameter to use for identifying which type of middleware is being registered.

options (*uri*, *host=None*, *strict_slashes=None*, *version=None*, *name=None*)

Add an API URL under the **OPTIONS HTTP** method

Parameters

- **uri** – URL to be tagged to **OPTIONS** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with `route()` method

patch (*uri*, *host=None*, *strict_slashes=None*, *stream=False*, *version=None*, *name=None*)

Add an API URL under the **DELETE HTTP** method

Parameters

- **uri** – URL to be tagged to **PATCH** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

post (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **POST** *HTTP* method

Parameters

- **uri** – URL to be tagged to **POST** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

put (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **PUT** *HTTP* method

Parameters

- **uri** – URL to be tagged to **PUT** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

register_blueprint (**args, **kwargs*)

Proxy method provided for invoking the *blueprint()* method

Note: To be deprecated in 1.0. Use *blueprint()* instead.

Parameters

- **args** – Blueprint object or (list, tuple) thereof
- **kwargs** – option dictionary with blueprint defaults

Returns None

register_listener (*listener, event*)

Register the listener for a given event.

Args: listener: callable i.e. `setup_db(app, loop)` event: when to register listener i.e. `'before_server_start'`

Returns: listener

register_middleware (*middleware, attach_to='request'*)

Register an application level middleware that will be attached to all the API URLs registered under this application.

This method is internally invoked by the `middleware()` decorator provided at the app level.

Parameters

- **middleware** – Callback method to be attached to the middleware
- **attach_to** – The state at which the middleware needs to be invoked in the lifecycle of an *HTTP Request*. **request** - Invoke before the request is processed **response** - Invoke before the response is returned back

Returns decorated method

remove_route (*uri, clean_cache=True, host=None*)

This method provides the app user a mechanism by which an already existing route can be removed from the *Sanic* object

Parameters

- **uri** – URL Path to be removed from the app
- **clean_cache** – Instruct sanic if it needs to clean up the LRU route cache
- **host** – IP address or FQDN specific to the host

Returns None

route (*uri, methods=frozenset({'GET'}), host=None, strict_slashes=None, stream=False, version=None, name=None*)

Decorate a function to be registered as a route

Parameters

- **uri** – path of the URL
- **methods** – list or tuple of methods allowed
- **host** –
- **strict_slashes** –
- **stream** –
- **version** –
- **name** – user defined route name for `url_for`

Returns decorated function

run (*host: Optional[str] = None, port: Optional[int] = None, debug: bool = False, ssl: Union[dict, ssl.SSLContext, None] = None, sock: Optional[socket.socket] = None, workers: int = 1, protocol: Type[asyncio.protocols.Protocol] = None, backlog: int = 100, stop_event: Any = None, register_sys_signals: bool = True, access_log: Optional[bool] = None, **kwargs*) → None

Run the HTTP Server and listen until keyboard interrupt or term signal. On termination, drain connections before closing.

Parameters

- **host** (*str*) – Address to host on
- **port** (*int*) – Port to host on
- **debug** (*bool*) – Enables debug output (slows server)
- **ssl** – SSLContext, or location of certificate and key for SSL encryption of worker(s)

:type ssl:SSLContext or dict :param sock: Socket for the server to accept connections from :type sock: socket :param workers: Number of processes received before it is respected :type workers: int :param protocol: Subclass of asyncio Protocol class :type protocol: type[Protocol] :param backlog: a number of unaccepted connections that the system

will allow before refusing new connections

Parameters

- **stop_event** (*None*) – event to be triggered before stopping the app - deprecated
- **register_sys_signals** (*bool*) – Register SIG* events
- **access_log** (*bool*) – Enables writing access logs (slows server)

Returns Nothing

static (*uri, file_or_directory, pattern='/?.+', use_modified_since=True, use_content_range=False, stream_large_files=False, name='static', host=None, strict_slashes=None, content_type=None*)

Register a root to serve files from. The input can either be a file or a directory. This method will enable an easy and simple way to setup the `Route` necessary to serve the static files.

Parameters

- **uri** – URL path to be used for serving static content
- **file_or_directory** – Path for the Static file/directory with static files
- **pattern** – Regex Pattern identifying the valid static files
- **use_modified_since** – If true, send file modified time, and return not modified if the browser's matches the server's
- **use_content_range** – If true, process header for range requests and sends the file part that is requested
- **stream_large_files** – If true, use the `StreamingHTTPResponse.file_stream()` handler rather than the `HTTPResponse.file()` handler to send the file. If this is an integer, this represents the threshold size to switch to `StreamingHTTPResponse.file_stream()`
- **name** – user defined name used for url_for
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *Sanic* to check if the request URLs need to terminate with a /
- **content_type** – user defined content type for header

Returns None

stop()

This kills the Sanic

test_client

coroutine trigger_events (*events, loop*)

Trigger events (functions or async) :param events: one or more sync or async functions to execute :param loop: event loop

url_for (*view_name: str, **kwargs*)

Build a URL based on a view name and the values provided.

In order to build a URL, all request parameters must be supplied as keyword arguments, and each parameter must pass the test for the specified parameter type. If these conditions are not met, a *URLBuildError* will be thrown.

Keyword arguments that are not request parameters will be included in the output URL's query string.

Parameters

- **view_name** – string referencing the view name
- ****kwargs** – keys and values that are used to build request parameters and query string arguments.

Returns the built URL

Raises: *URLBuildError*

websocket (*uri, host=None, strict_slashes=None, subprotocols=None, name=None*)

Decorate a function to be registered as a websocket route :param uri: path of the URL :param subprotocols: optional list of str with supported subprotocols :param host: :return: decorated function

class `sanic.Blueprint` (*name, url_prefix=None, host=None, version=None, strict_slashes=False*)

Bases: `object`

add_route (*handler, uri, methods=frozenset({'GET'}), host=None, strict_slashes=None, version=None, name=None, stream=False*)

Create a blueprint route from a function.

Parameters

- **handler** – function for handling uri requests. Accepts function, or class instance with a `view_class` method.
- **uri** – endpoint at which the route will be accessible.
- **methods** – list of acceptable HTTP methods.
- **host** – IP Address of FQDN for the sanic server to use.
- **strict_slashes** – Enforce the API urls are requested with a training /
- **version** – Blueprint Version
- **name** – user defined route name for `url_for`
- **stream** – boolean specifying if the handler is a stream handler

Returns function or class instance

add_websocket_route (*handler, uri, host=None, version=None, name=None*)

Create a blueprint websocket route from a function.

Parameters

- **handler** – function for handling uri requests. Accepts function, or class instance with a `view_class` method.
- **uri** – endpoint at which the route will be accessible.

- **host** – IP Address of FQDN for the sanic server to use.
- **version** – Blueprint Version
- **name** – Unique name to identify the Websocket Route

Returns function or class instance

delete (*uri, host=None, strict_slashes=None, version=None, name=None*)
Add an API URL under the **DELETE HTTP** method

Parameters

- **uri** – URL to be tagged to **DELETE** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

exception (**args, **kwargs*)

This method enables the process of creating a global exception handler for the current blueprint under question.

Parameters

- **args** – List of Python exceptions to be caught by the handler
- **kwargs** – Additional optional arguments to be passed to the exception handler

:return a decorated method to handle global exceptions for any route registered under this blueprint.

get (*uri, host=None, strict_slashes=None, version=None, name=None*)
Add an API URL under the **GET HTTP** method

Parameters

- **uri** – URL to be tagged to **GET** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

static group (**blueprints, url_prefix=""*)

Create a list of blueprints, optionally grouping them under a general URL prefix.

Parameters

- **blueprints** – blueprints to be registered as a group
- **url_prefix** – URL route to be prepended to all sub-prefixes

head (*uri, host=None, strict_slashes=None, version=None, name=None*)
Add an API URL under the **HEAD HTTP** method

Parameters

- **uri** – URL to be tagged to **HEAD** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

listener (*event*)

Create a listener from a decorated function.

Parameters **event** – Event to listen to.

middleware (**args, **kwargs*)

Create a blueprint middleware from a decorated function.

Parameters

- **args** – Positional arguments to be used while invoking the middleware
- **kwargs** – optional keyword args that can be used with the middleware.

options (*uri, host=None, strict_slashes=None, version=None, name=None*)

Add an API URL under the **OPTIONS HTTP** method

Parameters

- **uri** – URL to be tagged to **OPTIONS** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

patch (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **PATCH HTTP** method

Parameters

- **uri** – URL to be tagged to **PATCH** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

post (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)

Add an API URL under the **POST HTTP** method

Parameters

- **uri** – URL to be tagged to **POST** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

put (*uri, host=None, strict_slashes=None, stream=False, version=None, name=None*)
Add an API URL under the **PUT HTTP** method

Parameters

- **uri** – URL to be tagged to **PUT** method of *HTTP*
- **host** – Host IP or FQDN for the service to use
- **strict_slashes** – Instruct *sanic.app.Sanic* to check if the request URLs need to terminate with a /
- **version** – API Version
- **name** – Unique name that can be used to identify the Route

Returns Object decorated with *route()* method

register (*app, options*)
Register the blueprint to the *sanic* app.

Parameters

- **app** – Instance of *sanic.app.Sanic* class
- **options** – Options to be used while registering the blueprint into the app. *url_prefix* - URL Prefix to override the blueprint prefix

route (*uri, methods=frozenset({'GET'}), host=None, strict_slashes=None, stream=False, version=None, name=None*)
Create a blueprint route from a decorated function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **methods** – list of acceptable HTTP methods.
- **host** – IP Address of FQDN for the *sanic* server to use.
- **strict_slashes** – Enforce the API urls are requested with a training /
- **stream** – If the route should provide a streaming support
- **version** – Blueprint Version
- **name** – Unique name to identify the Route

:return a decorated method that when invoked will return an object of type *FutureRoute*

static (*uri, file_or_directory, *args, **kwargs*)
Create a blueprint static route from a decorated function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **file_or_directory** – Static asset.

websocket (*uri, host=None, strict_slashes=None, version=None, name=None*)

Create a blueprint websocket route from a decorated function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **host** – IP Address of FQDN for the sanic server to use.
- **strict_slashes** – Enforce the API urls are requested with a training /
- **version** – Blueprint Version
- **name** – Unique name to identify the Websocket Route

2.30 Python 3.7 AsyncIO examples

With Python 3.7 AsyncIO got major update for the following types:

- `asyncio.AbstractEventLoop`
- `asyncio.AnstractServer`

This example shows how to use sanic with Python 3.7, to be precise: how to retrieve an asyncio server instance:

```
import asyncio
import socket
import os

from sanic import Sanic
from sanic.response import json

app = Sanic(__name__)

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

server_socket = '/tmp/sanic.sock'

sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

try:
    os.remove(server_socket)
finally:
    sock.bind(server_socket)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    srv_coro = app.create_server(
        sock=sock,
        return_asyncio_server=True,
        asyncio_server_args=dict(
            start_serving=False
        )
    )
```

(continues on next page)

(continued from previous page)

```
    )
)
srv = loop.run_until_complete(srv_coro)
try:
    assert srv.is_serving() is False
    loop.run_until_complete(srv.start_serving())
    assert srv.is_serving() is True
    loop.run_until_complete(srv.serve_forever())
except KeyboardInterrupt:
    srv.close()
    loop.close()
```

Please note that uvloop does not support these features yet.

CHAPTER 3

Module Documentation

- [genindex](#)
- [modindex](#)
- [search](#)

S

- `sanic`, 100
- `sanic.app`, 70
- `sanic.blueprints`, 77
- `sanic.config`, 82
- `sanic.constants`, 83
- `sanic.cookies`, 83
- `sanic.exceptions`, 83
- `sanic.handlers`, 85
- `sanic.log`, 87
- `sanic.request`, 87
- `sanic.response`, 89
- `sanic.router`, 91
- `sanic.server`, 94
- `sanic.static`, 98
- `sanic.testing`, 98
- `sanic.views`, 99
- `sanic.websocket`, 100

A

abort() (in module *sanic.exceptions*), 85
 access_log (*sanic.server.HttpProtocol* attribute), 94
 add() (*sanic.handlers.ErrorHandler* method), 86
 add() (*sanic.router.Router* method), 92
 add() (*sanic.views.CompositionView* method), 99
 add_route() (*sanic.app.Sanic* method), 70
 add_route() (*sanic.Blueprint* method), 107
 add_route() (*sanic.blueprints.Blueprint* method), 77
 add_route() (*sanic.Sanic* method), 100
 add_status_code() (in module *sanic.exceptions*), 85
 add_task() (*sanic.app.Sanic* method), 71
 add_task() (*sanic.Sanic* method), 101
 add_websocket_route() (*sanic.app.Sanic* method), 71
 add_websocket_route() (*sanic.Blueprint* method), 107
 add_websocket_route() (*sanic.blueprints.Blueprint* method), 77
 add_websocket_route() (*sanic.Sanic* method), 101
 app (*sanic.request.Request* attribute), 87
 args (*sanic.blueprints.FutureException* attribute), 81
 args (*sanic.blueprints.FutureMiddleware* attribute), 81
 args (*sanic.blueprints.FutureStatic* attribute), 82
 args (*sanic.request.Request* attribute), 87
 as_view() (*sanic.views.HTTPMethodView* class method), 99

B

bail_out() (*sanic.server.HttpProtocol* method), 94
 BaseHTTPResponse (class in *sanic.response*), 89
 Blueprint (class in *sanic*), 107
 Blueprint (class in *sanic.blueprints*), 77
 blueprint() (*sanic.app.Sanic* method), 71
 blueprint() (*sanic.Sanic* method), 101
 body (*sanic.request.File* attribute), 87
 body (*sanic.request.Request* attribute), 87

body (*sanic.response.HTTPResponse* attribute), 89
 body_append() (*sanic.server.HttpProtocol* method), 94
 body_finish() (*sanic.request.Request* method), 87
 body_init() (*sanic.request.Request* method), 87
 body_push() (*sanic.request.Request* method), 87

C

cached_handlers (*sanic.handlers.ErrorHandler* attribute), 86
 cast (*sanic.router.Parameter* attribute), 91
 check_dynamic_route_exists() (*sanic.router.Router* static method), 92
 cleanup() (*sanic.server.HttpProtocol* method), 94
 close() (*sanic.server.HttpProtocol* method), 94
 close_if_idle() (*sanic.server.HttpProtocol* method), 94
 CompositionView (class in *sanic.views*), 99
 Config (class in *sanic.config*), 82
 connection_lost() (*sanic.server.HttpProtocol* method), 94
 connection_lost() (*sanic.websocket.WebSocketProtocol* method), 100
 connection_made() (*sanic.server.HttpProtocol* method), 94
 connections (*sanic.server.HttpProtocol* attribute), 94
 content_type (*sanic.request.Request* attribute), 87
 content_type (*sanic.response.HTTPResponse* attribute), 89
 content_type (*sanic.response.StreamingHTTPResponse* attribute), 89
 ContentRangeError, 83
 ContentRangeHandler (class in *sanic.handlers*), 85
 converted_response_type() (*sanic.app.Sanic* method), 71
 converted_response_type() (*sanic.Sanic* method), 101
 Cookie (class in *sanic.cookies*), 83
 CookieJar (class in *sanic.cookies*), 83

cookies (*sanic.request.Request attribute*), 87
 cookies (*sanic.response.BaseHTTPResponse attribute*), 89
 cookies (*sanic.response.HTTPResponse attribute*), 89
 create_server() (*sanic.app.Sanic method*), 71
 create_server() (*sanic.Sanic method*), 101

D

data_received() (*sanic.server.HttpProtocol method*), 94
 data_received() (*sanic.websocket.WebSocketProtocol method*), 100
 decorators (*sanic.views.HTTPMethodView attribute*), 99
 default() (*sanic.handlers.ErrorHandler method*), 86
 delete() (*sanic.app.Sanic method*), 72
 delete() (*sanic.Blueprint method*), 108
 delete() (*sanic.blueprints.Blueprint method*), 78
 delete() (*sanic.Sanic method*), 102
 delete() (*sanic.testing.SanicTestClient method*), 98
 dispatch_request() (*sanic.views.HTTPMethodView method*), 99
 drain() (*sanic.server.HttpProtocol method*), 94

E

enable_websocket() (*sanic.app.Sanic method*), 72
 enable_websocket() (*sanic.Sanic method*), 102
 encode() (*sanic.cookies.Cookie method*), 83
 end (*sanic.handlers.ContentRangeHandler attribute*), 85
 endpoint (*sanic.request.Request attribute*), 87
 error_handler (*sanic.server.HttpProtocol attribute*), 94
 ErrorHandler (*class in sanc.handlers*), 86
 exception() (*sanic.app.Sanic method*), 72
 exception() (*sanic.Blueprint method*), 108
 exception() (*sanic.blueprints.Blueprint method*), 78
 exception() (*sanic.Sanic method*), 102
 execute_request_handler() (*sanic.server.HttpProtocol method*), 94

F

File (*class in sanc.request*), 87
 file() (*in module sanc.response*), 89
 file_or_directory (*sanic.blueprints.FutureStatic attribute*), 82
 file_stream() (*in module sanc.response*), 89
 FileNotFoundError, 83
 files (*sanic.request.Request attribute*), 87
 find_route_by_view_name (*sanic.router.Router attribute*), 93
 Forbidden, 83
 form (*sanic.request.Request attribute*), 87
 from_envvar() (*sanic.config.Config method*), 82

from_object() (*sanic.config.Config method*), 82
 from_pyfile() (*sanic.config.Config method*), 82
 FutureException (*class in sanc.blueprints*), 81
 FutureListener (*class in sanc.blueprints*), 81
 FutureMiddleware (*class in sanc.blueprints*), 81
 FutureRoute (*class in sanc.blueprints*), 81
 FutureStatic (*class in sanc.blueprints*), 82

G

get() (*sanic.app.Sanic method*), 72
 get() (*sanic.Blueprint method*), 108
 get() (*sanic.blueprints.Blueprint method*), 78
 get() (*sanic.request.RequestParameters method*), 88
 get() (*sanic.router.Router method*), 93
 get() (*sanic.Sanic method*), 102
 get() (*sanic.testing.SanicTestClient method*), 98
 get_headers() (*sanic.response.StreamingHTTPResponse method*), 89
 get_supported_methods() (*sanic.router.Router method*), 93
 getlist() (*sanic.request.RequestParameters method*), 88
 group() (*sanic.Blueprint static method*), 108
 group() (*sanic.blueprints.Blueprint static method*), 78

H

handle_request() (*sanic.app.Sanic method*), 73
 handle_request() (*sanic.Sanic method*), 103
 handler (*sanic.blueprints.FutureException attribute*), 81
 handler (*sanic.blueprints.FutureListener attribute*), 81
 handler (*sanic.blueprints.FutureRoute attribute*), 81
 handler (*sanic.router.Route attribute*), 91
 handlers (*sanic.handlers.ErrorHandler attribute*), 86
 head() (*sanic.app.Sanic method*), 73
 head() (*sanic.Blueprint method*), 108
 head() (*sanic.blueprints.Blueprint method*), 78
 head() (*sanic.Sanic method*), 103
 head() (*sanic.testing.SanicTestClient method*), 98
 HeaderNotFound, 83
 headers (*sanic.handlers.ContentRangeHandler attribute*), 85
 headers (*sanic.request.Request attribute*), 87
 headers (*sanic.response.HTTPResponse attribute*), 89
 headers (*sanic.response.StreamingHTTPResponse attribute*), 89
 headers (*sanic.server.HttpProtocol attribute*), 95
 host (*sanic.blueprints.FutureListener attribute*), 81
 host (*sanic.blueprints.FutureRoute attribute*), 81
 host (*sanic.request.Request attribute*), 87
 html() (*in module sanc.response*), 90
 HTTPMethodView (*class in sanc.views*), 99
 HttpProtocol (*class in sanc.server*), 94
 HTTPResponse (*class in sanc.response*), 89

I

InvalidRangeType, 83
 InvalidUsage, 83
 ip (*sanic.request.Request* attribute), 87
 is_full() (*sanic.request.StreamBuffer* method), 88
 is_request_stream (*sanic.server.HttpProtocol* attribute), 95
 is_stream_handler() (*sanic.router.Router* method), 93

J

json (*sanic.request.Request* attribute), 87
 json() (in module *sanic.response*), 90

K

keep_alive (*sanic.server.HttpProtocol* attribute), 95
 keep_alive_timeout (*sanic.server.HttpProtocol* attribute), 95
 keep_alive_timeout_callback() (*sanic.server.HttpProtocol* method), 95
 keep_alive_timeout_callback() (*sanic.websocket.WebSocketProtocol* method), 100
 kwargs (*sanic.blueprints.FutureException* attribute), 81
 kwargs (*sanic.blueprints.FutureMiddleware* attribute), 81
 kwargs (*sanic.blueprints.FutureStatic* attribute), 82

L

listener() (*sanic.app.Sanic* method), 73
 listener() (*sanic.Blueprint* method), 109
 listener() (*sanic.blueprints.Blueprint* method), 79
 listener() (*sanic.Sanic* method), 103
 load_environment_vars() (*sanic.config.Config* method), 82
 load_json() (*sanic.request.Request* method), 87
 log() (*sanic.handlers.ErrorHandler* method), 86
 log_response() (*sanic.server.HttpProtocol* method), 95
 lookup() (*sanic.handlers.ErrorHandler* method), 86
 loop (*sanic.app.Sanic* attribute), 73
 loop (*sanic.Sanic* attribute), 103
 loop (*sanic.server.HttpProtocol* attribute), 95

M

match_info (*sanic.request.Request* attribute), 87
 method (*sanic.request.Request* attribute), 87
 MethodNotSupported, 83
 methods (*sanic.blueprints.FutureListener* attribute), 81
 methods (*sanic.blueprints.FutureRoute* attribute), 81
 methods (*sanic.router.Route* attribute), 91
 middleware (*sanic.blueprints.FutureMiddleware* attribute), 81

middleware() (*sanic.app.Sanic* method), 73
 middleware() (*sanic.Blueprint* method), 109
 middleware() (*sanic.blueprints.Blueprint* method), 79
 middleware() (*sanic.Sanic* method), 103

N

name (*sanic.blueprints.FutureRoute* attribute), 81
 name (*sanic.request.File* attribute), 87
 name (*sanic.router.Parameter* attribute), 91
 name (*sanic.router.Route* attribute), 91
 NotFound, 83

O

on_body() (*sanic.server.HttpProtocol* method), 95
 on_header() (*sanic.server.HttpProtocol* method), 95
 on_headers_complete() (*sanic.server.HttpProtocol* method), 95
 on_message_complete() (*sanic.server.HttpProtocol* method), 95
 on_url() (*sanic.server.HttpProtocol* method), 95
 options() (*sanic.app.Sanic* method), 73
 options() (*sanic.Blueprint* method), 109
 options() (*sanic.blueprints.Blueprint* method), 79
 options() (*sanic.Sanic* method), 103
 options() (*sanic.testing.SanicTestClient* method), 98
 output() (*sanic.response.HTTPResponse* method), 89

P

Parameter (class in *sanic.router*), 91
 parameter_pattern (*sanic.router.Router* attribute), 93
 ParameterNameConflicts, 91
 parameters (*sanic.router.Route* attribute), 92
 parse_multipart_form() (in module *sanic.request*), 88
 parse_parameter_string() (*sanic.router.Router* class method), 93
 parsed_args (*sanic.request.Request* attribute), 87
 parsed_files (*sanic.request.Request* attribute), 87
 parsed_form (*sanic.request.Request* attribute), 88
 parsed_json (*sanic.request.Request* attribute), 88
 parser (*sanic.server.HttpProtocol* attribute), 95
 patch() (*sanic.app.Sanic* method), 73
 patch() (*sanic.Blueprint* method), 109
 patch() (*sanic.blueprints.Blueprint* method), 79
 patch() (*sanic.Sanic* method), 103
 patch() (*sanic.testing.SanicTestClient* method), 99
 path (*sanic.request.Request* attribute), 88
 pattern (*sanic.router.Route* attribute), 92
 pause_writing() (*sanic.server.HttpProtocol* method), 95
 PayloadTooLarge, 84
 port (*sanic.request.Request* attribute), 88

post () (*sanic.app.Sanic method*), 74
 post () (*sanic.Blueprint method*), 109
 post () (*sanic.blueprints.Blueprint method*), 79
 post () (*sanic.Sanic method*), 104
 post () (*sanic.testing.SanicTestClient method*), 99
 protocol (*sanic.response.StreamingHTTPResponse attribute*), 89
 push_data () (*sanic.server.HttpProtocol method*), 95
 put () (*sanic.app.Sanic method*), 74
 put () (*sanic.Blueprint method*), 110
 put () (*sanic.blueprints.Blueprint method*), 80
 put () (*sanic.request.StreamBuffer method*), 88
 put () (*sanic.Sanic method*), 104
 put () (*sanic.testing.SanicTestClient method*), 99
 PyFileError, 84

Q

query_string (*sanic.request.Request attribute*), 88

R

raw () (*in module sancic.response*), 90
 raw_args (*sanic.request.Request attribute*), 88
 raw_url (*sanic.request.Request attribute*), 88
 read () (*sanic.request.StreamBuffer method*), 88
 redirect () (*in module sancic.response*), 90
 register () (*in module sancic.static*), 98
 register () (*sanic.Blueprint method*), 110
 register () (*sanic.blueprints.Blueprint method*), 80
 register_blueprint () (*sanic.app.Sanic method*), 74
 register_blueprint () (*sanic.Sanic method*), 104
 register_listener () (*sanic.app.Sanic method*), 74
 register_listener () (*sanic.Sanic method*), 104
 register_middleware () (*sanic.app.Sanic method*), 74
 register_middleware () (*sanic.Sanic method*), 105
 remote_addr (*sanic.request.Request attribute*), 88
 remove () (*sanic.router.Router method*), 93
 remove_route () (*sanic.app.Sanic method*), 75
 remove_route () (*sanic.Sanic method*), 105
 Request (*class in sancic.request*), 87
 request (*sanic.server.HttpProtocol attribute*), 95
 request_buffer_queue_size (*sanic.server.HttpProtocol attribute*), 95
 request_class (*sanic.server.HttpProtocol attribute*), 95
 request_handler (*sanic.server.HttpProtocol attribute*), 96
 request_max_size (*sanic.server.HttpProtocol attribute*), 96
 request_timeout (*sanic.server.HttpProtocol attribute*), 96

request_timeout_callback () (*sanic.server.HttpProtocol method*), 96
 request_timeout_callback () (*sanic.websocket.WebSocketProtocol method*), 100
 RequestParameters (*class in sancic.request*), 88
 RequestTimeout, 84
 response () (*sanic.handlers.ErrorHandler method*), 86
 response_timeout (*sanic.server.HttpProtocol attribute*), 96
 response_timeout_callback () (*sanic.server.HttpProtocol method*), 96
 response_timeout_callback () (*sanic.websocket.WebSocketProtocol method*), 100
 resume_writing () (*sanic.server.HttpProtocol method*), 96
 Route (*class in sancic.router*), 91
 route () (*sanic.app.Sanic method*), 75
 route () (*sanic.Blueprint method*), 110
 route () (*sanic.blueprints.Blueprint method*), 80
 route () (*sanic.Sanic method*), 105
 RouteDoesNotExist, 92
 RouteExists, 92
 Router (*class in sancic.router*), 92
 router (*sanic.server.HttpProtocol attribute*), 96
 routes_always_check (*sanic.router.Router attribute*), 93
 routes_dynamic (*sanic.router.Router attribute*), 93
 routes_static (*sanic.router.Router attribute*), 93
 run () (*sanic.app.Sanic method*), 75
 run () (*sanic.Sanic method*), 105

S

Sanic (*class in sancic*), 100
 Sanic (*class in sancic.app*), 70
 sancic (*module*), 100
 sancic.app (*module*), 70
 sancic.blueprints (*module*), 77
 sancic.config (*module*), 82
 sancic.constants (*module*), 83
 sancic.cookies (*module*), 83
 sancic.exceptions (*module*), 83
 sancic.handlers (*module*), 85
 sancic.log (*module*), 87
 sancic.request (*module*), 87
 sancic.response (*module*), 89
 sancic.router (*module*), 91
 sancic.server (*module*), 94
 sancic.static (*module*), 98
 sancic.testing (*module*), 98
 sancic.views (*module*), 99
 sancic.websocket (*module*), 100

- SanicException, 84
- SanicTestClient (*class in sanic.testing*), 98
- scheme (*sanic.request.Request attribute*), 88
- serve() (*in module sanic.server*), 96
- serve_multiple() (*in module sanic.server*), 97
- ServerError, 84
- ServiceUnavailable, 84
- Signal (*class in sanic.server*), 96
- signal (*sanic.server.HttpProtocol attribute*), 96
- size (*sanic.handlers.ContentRangeHandler attribute*), 85
- socket (*sanic.request.Request attribute*), 88
- start (*sanic.handlers.ContentRangeHandler attribute*), 86
- state (*sanic.server.HttpProtocol attribute*), 96
- static() (*sanic.app.Sanic method*), 76
- static() (*sanic.Blueprint method*), 110
- static() (*sanic.blueprints.Blueprint method*), 80
- static() (*sanic.Sanic method*), 106
- status (*sanic.response.HTTPResponse attribute*), 89
- status (*sanic.response.StreamingHTTPResponse attribute*), 89
- status_code (*sanic.exceptions.ContentRangeError attribute*), 83
- status_code (*sanic.exceptions.Forbidden attribute*), 83
- status_code (*sanic.exceptions.InvalidUsage attribute*), 83
- status_code (*sanic.exceptions.MethodNotSupported attribute*), 83
- status_code (*sanic.exceptions.NotFound attribute*), 84
- status_code (*sanic.exceptions.PayloadTooLarge attribute*), 84
- status_code (*sanic.exceptions.RequestTimeout attribute*), 84
- status_code (*sanic.exceptions.ServerError attribute*), 84
- status_code (*sanic.exceptions.ServiceUnavailable attribute*), 84
- status_code (*sanic.exceptions.Unauthorized attribute*), 85
- stop() (*sanic.app.Sanic method*), 76
- stop() (*sanic.Sanic method*), 106
- stopped (*sanic.server.Signal attribute*), 96
- stream (*sanic.blueprints.FutureRoute attribute*), 82
- stream (*sanic.request.Request attribute*), 88
- stream() (*in module sanic.response*), 91
- stream() (*in module sanic.views*), 100
- stream() (*sanic.response.StreamingHTTPResponse method*), 89
- stream_response() (*sanic.server.HttpProtocol method*), 96
- StreamBuffer (*class in sanic.request*), 88
- streaming_fn (*sanic.response.StreamingHTTPResponse attribute*), 89
- StreamingHTTPResponse (*class in sanic.response*), 89
- strict_slashes (*sanic.blueprints.FutureRoute attribute*), 82
- ## T
- test_client (*sanic.app.Sanic attribute*), 76
- test_client (*sanic.Sanic attribute*), 106
- text() (*in module sanic.response*), 91
- token (*sanic.request.Request attribute*), 88
- total (*sanic.handlers.ContentRangeHandler attribute*), 86
- transport (*sanic.request.Request attribute*), 88
- transport (*sanic.server.HttpProtocol attribute*), 96
- trigger_events() (*in module sanic.server*), 98
- trigger_events() (*sanic.app.Sanic method*), 76
- trigger_events() (*sanic.Sanic method*), 106
- type (*sanic.request.File attribute*), 87
- ## U
- Unauthorized, 84
- update_current_time() (*in module sanic.server*), 98
- uri (*sanic.blueprints.FutureListener attribute*), 81
- uri (*sanic.blueprints.FutureRoute attribute*), 82
- uri (*sanic.blueprints.FutureStatic attribute*), 82
- uri (*sanic.router.Route attribute*), 92
- uri_template (*sanic.request.Request attribute*), 88
- url (*sanic.request.Request attribute*), 88
- url (*sanic.server.HttpProtocol attribute*), 96
- url_for() (*sanic.app.Sanic method*), 76
- url_for() (*sanic.Sanic method*), 107
- url_hash() (*in module sanic.router*), 93
- URLBuildError, 84
- ## V
- version (*sanic.blueprints.FutureRoute attribute*), 82
- version (*sanic.request.Request attribute*), 88
- ## W
- websocket() (*sanic.app.Sanic method*), 77
- websocket() (*sanic.Blueprint method*), 111
- websocket() (*sanic.blueprints.Blueprint method*), 80
- websocket() (*sanic.Sanic method*), 107
- websocket_handshake() (*sanic.websocket.WebSocketProtocol method*), 100
- WebSocketProtocol (*class in sanic.websocket*), 100
- write() (*sanic.response.StreamingHTTPResponse method*), 89
- write_error() (*sanic.server.HttpProtocol method*), 96

`write_response()` (*sanic.server.HttpProtocol*
method), 96

`write_response()` (*sanic.websocket.WebSocketProtocol*
method), 100