Sanic Documentation

Release 0.7.0

Sanic contributors

Dec 26, 2017

Contents

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)
```

CHAPTER 2

Guides

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.

1. Install Sanic: `python3 -m pip install sanic`
2. Create a file called `main.py` with the following code:

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

app = Sanic()

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

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

1. Run the server: `python3 main.py`
2. 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 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.

2.2.1 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))
```

2.2.2 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 a also a route with no host, it will be the default.

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

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

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))
```

2.2.3 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'])
```

2.2.4 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:

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


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

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 support 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 passed argument _server or SERVER_NAME in app.config or url will␣
↪be same as no _external

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

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

- 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 thrown.

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

WebSocket support requires the websockets package by Aymeric Augustin.

2.2.6 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)
```

2.2.7 User defined route name

You can pass `name` to change the route name to avoid using the default name (`handler.__name__`).

```

app = Sanic('test_named_route')

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

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

```
# instead of # `app.url_for('handler')`

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

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

app.blueprint(bp)

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

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

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

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

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

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

2.2.8 Build URL for static files

You can use `url_for` for static file url building now. If it's for file directly, `filename` can be ignored.

```
app = Sanic('test_static')
app.static('/static', './static')
app.static('/uploads', './uploads', name='uploads')
```

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

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

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

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

2.3 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.
- `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=`

2.3.1 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 Response

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

2.4.1 Plain Text

```
from sanic import response


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

2.4.2 HTML

```
from sanic import response


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

2.4.3 JSON

```
from sanic import response


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

2.4.4 File

```
from sanic import response


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

2.4.5 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')
```

2.4.6 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')
```

2.4.7 Redirect

```
from sanic import response


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

2.4.8 Raw

Response without encoding the body

```
from sanic import response


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

2.4.9 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
    )
```

2.5 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')
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)
```

2.6 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.

2.6.1 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')
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')
def no_no(request):
        abort(401)
        # this won't happen
        text("OK")
```

2.6.2 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)
```

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

2.6.3 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.

2.7 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.

2.7.1 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'`. Response middleware receives both the request and the response as arguments.

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")
```

2.7.2 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('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 two middleware in order. First, the middleware **custom_banner** will change the HTTP response header *Server* to *Fake-Server*, and the second 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.

2.7.3 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')
```

2.7.4 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')
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()
```

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())
```

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=None, pattern=re.compile(
↪'^/$'), parameters=[])]
```

2.8.3 Using blueprints

Blueprints have much 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')
# 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'
```

2.8.4 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()
```

2.8.5 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 `url_prefix` 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='/v1')
blueprint_v2 = Blueprint('v2', url_prefix='/v2')

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

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

When we register our blueprints on the app, the routes `/v1` and `/v2` 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, url_prefix='/v1')
app.blueprint(blueprint_v2, url_prefix='/v2')

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

2.8.6 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/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 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.

2.9.1 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.

2.9.2 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'
```

2.9.3 Builtin Configuration Values

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

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

The different Timeout variables:

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

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

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

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

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

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

For reference:

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

2.10 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.

2.10.1 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))
```

2.10.2 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
```

2.10.3 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.

2.11 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.

2.11.1 Authorization Decorator

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

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

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

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


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

2.12 Streaming

2.12.1 Request Streaming

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

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

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


class SimpleView(HTTPMethodView):

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


@app.post('/stream', stream=True)
async def handler(request):
    async def streaming(response):
        while True:
            body = await request.stream.get()
            if body is None:
                break
            body = body.decode('utf-8').replace('1', 'A')
            response.write(body)
    return stream(streaming)


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


async def post_handler(request):
    result = ''
    while True:
```

```
        body = await request.stream.get()
        if body is None:
            break
        result += body.decode('utf-8')
    return text(result)

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


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

2.12.2 Response Streaming

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

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

app = Sanic(__name__)

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

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

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

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

    return stream(stream_from_db)
```

2.13 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.

2.13.1 Defining views

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

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

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

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

app = Sanic('some_name')

class SimpleView(HTTPMethodView):

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

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

  def put(self, request):
      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(), '/')
```

2.13.2 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>')
```

2.13.3 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')

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

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')
```

2.13.4 Using CompositionView

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

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

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

app = Sanic(__name__)

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

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`.

2.14 Custom Protocols

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

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

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

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

2.14.1 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__)
```

```
class CustomHttpProtocol(HttpProtocol):

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

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


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


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

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

2.15 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)
```

2.16 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.

2.16.1 Quick Start

A simple example using default settings would be like this:

```
from sanic import Sanic

app = Sanic('test')

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

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

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

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

And to close logging, simply assign access_log=False:

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

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

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

2.16.2 Configuration

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

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

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

Log format:

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

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

The default access log format is

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

2.17 Testing

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

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

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

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

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

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

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

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

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

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

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

And to supply data to a JSON POST request:

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

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

2.17.1 pytest-sanic

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

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

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

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

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

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

    yield app


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


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

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}
```

2.18 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.

2.18.1 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.

2.18.2 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)
```

2.18.3 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.

2.18.4 Asynchronous support

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

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

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

2.19 Extensions

A list of Sanic extensions created by the community.

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

- sanic-prometheus: Prometheus metrics for Sanic
- Sanic-RestPlus: A port of Flask-RestPlus for Sanic. Full-featured REST API with SwaggerUI generation.
- sanic-transmute: A Sanic extension that generates APIs from python function and classes, and also generates Swagger UI/documentation automatically.
- pytest-sanic: A pytest plugin for Sanic. It helps you to test your code asynchronously.
- jinja2-sanic: a jinja2 template renderer for Sanic.(Documentation)

2.20 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.

2.20.1 Installation

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

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

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

2.20.2 Running tests

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

```
tox
```

See it's that simple!

2.20.3 Pull requests!

So the pull request approval rules are pretty simple:

1. All pull requests must pass unit tests

- All pull requests must be reviewed and approved by at least one current collaborator on the project
- All pull requests must pass flake8 checks
- If you decide to remove/change anything from any common interface a deprecation message should accompany it.
- If you implement a new feature you should have at least one unit test to accompany it.

2.20.4 Documentation

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

To generate the documentation from scratch:

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

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

2.20.5 Warning

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

2.21 API Reference

2.21.1 Submodules

2.21.2 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*)

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

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, couroutine or awaitable

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

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

blueprint(*blueprint*, ***options*)

Register a blueprint on the application.

Parameters

- **blueprint** – Blueprint object
- **options** – option dictionary with blueprint defaults

Returns Nothing

converted_response_type(*response*)

coroutine create_server(*host=None*, *port=None*, *debug=False*, *ssl=None*, *sock=None*, *protocol=None*, *backlog=100*, *stop_event=None*, *access_log=True*)

Asynchronous version of *run*.

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

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

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*)

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')

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

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

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

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

register_blueprint(**args*, ***kwargs*)

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

remove_route(*uri*, *clean_cache=True*, *host=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=None*, *port=None*, *debug=False*, *ssl=None*, *sock=None*, *workers=1*, *protocol=None*, *backlog=100*, *stop_event=None*, *register_sys_signals=True*, *access_log=True*)
Run the HTTP Server and listen until keyboard interrupt or term signal. On termination, drain connections before closing.

Parameters

- **host** – Address to host on
- **port** – Port to host on
- **debug** – Enables debug output (slows server)
- **ssl** – SSLContext, or location of certificate and key for SSL encryption of worker(s)
- **sock** – Socket for the server to accept connections from
- **workers** – Number of processes received before it is respected
- **backlog** –
- **stop_event** –
- **register_sys_signals** –
- **protocol** – Subclass of asyncio protocol class

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*)
Register a root to serve files from. The input can either be a file or a directory. See

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 strings with the supported

subprotocols

Parameters **host** –

Returns decorated function

2.21.3 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*)
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** –
- **strict_slashes** –
- **version** –
- **name** – user defined route name for url_for

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.

Returns function or class instance

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

exception(**args*, ***kwargs*)
Create a blueprint exception from a decorated function.

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

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.

middleware(**args*, ***kwargs*)
Create a blueprint middleware from a decorated function.

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

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

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

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

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

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.

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.

sanic.blueprints.**FutureException**
alias of `Route`

sanic.blueprints.**FutureListener**
alias of Listener

sanic.blueprints.**FutureMiddleware**
alias of Route

sanic.blueprints.**FutureRoute**
alias of Route

sanic.blueprints.**FutureStatic**
alias of Route

2.21.4 sanic.config module

class sanic.config.**Config**(*defaults=None*, *load_env=True*, *keep_alive=True*)
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.

2.21.5 sanic.constants module

2.21.6 sanic.cookies module

class sanic.cookies.**Cookie**(*key*, *value*)
Bases: dict

A stripped down version of Morsel from SimpleCookie #gottagofast

encode(*encoding*)

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.

class sanic.cookies.**MultiHeader**(*name*)
Bases: object

String-holding object which allow us to set a header within response that has a unique key, but may contain duplicate header names

encode()

2.21.7 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.**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")

# 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 _sanic_exceptions.

2.21.8 sanic.handlers module

class sanic.handlers.**ContentRangeHandler**(*request*, *stats*)
Bases: object

Class responsible for parsing request header

end

headers

size

start

total

class sanic.handlers.**ErrorHandler**
Bases: object

add(*exception*, *handler*)

cached_handlers = None

default(*request*, *exception*)

handlers = None

log(*message*, *level='error'*)
Override this method in an ErrorHandler subclass to prevent logging exceptions.

lookup(*exception*)

response(*request*, *exception*)
Fetches and executes an exception handler and returns a response object

Parameters

- **request** – Request
- **exception** – Exception to handle

Returns Response object

2.21.9 sanic.log module

2.21.10 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

content_type

cookies

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

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

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)

2.21.11 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

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

transport

write(*data*)
Writes a chunk of data to the streaming response.

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

coroutine sanic.response.**file**(*location*, *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*, *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

2.21.12 sanic.router module

class sanic.router.**Parameter**(*name*, *cast*)
Bases: tuple

cast
Alias for field number 1

name
Alias for field number 0

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*)

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*)

2.21.13 sanic.server module

class sanic.server.**CIDict**
Bases: dict

Case Insensitive dict where all keys are converted to lowercase This does not maintain the inputted case when calling items() or keys() in favor of speed, since headers are case insensitive

get(*key*, *default=None*)

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

access_log

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

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*)

connection_made(*transport*)

connections

data_received(*data*)

execute_request_handler()

headers

is_request_stream

keep_alive

keep_alive_timeout

keep_alive_timeout_callback()

log_response(*response*)

loop

on_body(*body*)

on_header(*name*, *value*)

on_headers_complete()

on_message_complete()

on_url(*url*)

parser

request

request_class

request_handler

request_max_size

request_timeout

request_timeout_callback()

response_timeout

response_timeout_callback()

router

signal

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*, *reuse_port=False*, *loop=None*, *protocol=<class 'sanic.server.HttpProtocol'>*, *backlog=100*, *register_sys_signals=True*, *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*, *state=None*, *graceful_shutdown_timeout=15.0*)
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
- **is_request_stream** – disable/enable Request.stream
- **router** – Router object

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

2.21.14 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*)

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

2.21.15 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*)

2.21.16 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*)

2.21.17 sanic.websocket module

class sanic.websocket.**WebSocketProtocol**(**args*, *websocket_max_size=None*, *websocket_max_queue=None*, ***kwargs*)
Bases: *sanic.server.HttpProtocol*

connection_lost(*exc*)

data_received(*data*)

keep_alive_timeout_callback()

request_timeout_callback()

response_timeout_callback()

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

write_response(*response*)

2.21.18 sanic.worker module

2.21.19 Module contents

class sanic.**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*)

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

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, couroutine or awaitable

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

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

blueprint(*blueprint*, ***options*)

Register a blueprint on the application.

Parameters

- **blueprint** – Blueprint object
- **options** – option dictionary with blueprint defaults

Returns Nothing

converted_response_type(*response*)

coroutine create_server(*host=None*, *port=None*, *debug=False*, *ssl=None*, *sock=None*, *protocol=None*, *backlog=100*, *stop_event=None*, *access_log=True*)

Asynchronous version of *run*.

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

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

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*)

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')

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

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

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

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

register_blueprint(**args*, ***kwargs*)

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

remove_route(*uri*, *clean_cache=True*, *host=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=None*, *port=None*, *debug=False*, *ssl=None*, *sock=None*, *workers=1*, *protocol=None*, *backlog=100*, *stop_event=None*, *register_sys_signals=True*, *access_log=True*)
Run the HTTP Server and listen until keyboard interrupt or term signal. On termination, drain connections before closing.

Parameters

- **host** – Address to host on
- **port** – Port to host on
- **debug** – Enables debug output (slows server)
- **ssl** – SSLContext, or location of certificate and key for SSL encryption of worker(s)
- **sock** – Socket for the server to accept connections from
- **workers** – Number of processes received before it is respected
- **backlog** –
- **stop_event** –
- **register_sys_signals** –
- **protocol** – Subclass of asyncio protocol class

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*)
Register a root to serve files from. The input can either be a file or a directory. See

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 strings with the supported

subprotocols

Parameters **host** –

Returns 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*)
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** –
- **strict_slashes** –
- **version** –
- **name** – user defined route name for url_for

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.

Returns function or class instance

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

exception(**args*, ***kwargs*)
Create a blueprint exception from a decorated function.

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

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.

middleware(**args*, ***kwargs*)
Create a blueprint middleware from a decorated function.

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

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

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

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

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

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.

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.

CHAPTER 3

Module Documentation

- genindex
- modindex
- search

Python Module Index

S

Index

M

N

O

P

Q

R

S

T

U

V

W