Sanic Documentation

Release 0.5.4

Sanic contributors

Aug 04, 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:

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

app = Sanic(__name__)

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

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

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:

```python
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.

```python
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.

```python
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.

```python
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.

2.3 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)
```

- `ip` (str) - IP address 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>/`

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 Redirect

```
from sanic import response


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

2.4.7 Raw

Response without encoding the body

```
from sanic import response


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

2.4.8 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
app = Sanic(__name__)

# Serves files from the static folder to the URL /static
app.static('/static', './static')

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

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

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

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)
```

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 providers 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.

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

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 = app.url_for('v1.post_handler', post_id=5) # --> '/v1/post/5'
    return redirect(url)


@blueprint_v1.route('/post/<post_id>')
```

```python
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:

```python
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:

```python
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. You can pass the `load_vars` boolean to the Sanic constructor to override that:

```python
app = Sanic(load_vars=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:

```python
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_file(/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 varibales 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 (sec) |
| KEEP_ALIVE        | True      | Disables keep-alive when False    |
```

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.

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 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:

```python
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:

```python
@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)
```

2.12 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.

2.12.1 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.

```python
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(), '/')
```

2.12.2 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>')
```

2.12.3 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')
```

2.12.4 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`.

2.13 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.

2.13.1 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)
```

2.14 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)
```

2.15 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 do create a new configuration.

2.15.1 Quick Start

A simple example using default setting would be like this:

```
from sanic import Sanic
from sanic.config import LOGGING

# The default logging handlers are ['accessStream', 'errorStream']
# but we change it to use other handlers here for demo purpose
LOGGING['loggers']['network']['handlers'] = [
    'accessTimedRotatingFile', 'errorTimedRotationgFile']

app = Sanic('test')

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

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

After the program starts, it will log down all the information/requests in access.log and error.log in your working directory.

And to close logging, simply assign log_config=None:

```
if __name__ == "__main__":
  app.run(log_config=None)
```

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 internal messages
  app.run(debug=False, log_config=None)
```

2.15.2 Configuration

By default, log_config parameter is set to use sanic.config.LOGGING dictionary for configuration. The default configuration provides several predefined `handlers`:

- internal (using logging.StreamHandler) For internal information console outputs.
- accessStream (using logging.StreamHandler) For requests information logging in console
- errorStream (using logging.StreamHandler) For error message and traceback logging in console.
- accessSysLog (using logging.handlers.SysLogHandler) For requests information logging to syslog. Currently supports Windows (via localhost:514), Darwin (/var/run/syslog), Linux (/dev/log) and FreeBSD (/dev/log). You would not be able to access this property if the directory doesn't exist. (Notice that in Docker you have to enable everything by yourself)
- errorSysLog (using logging.handlers.SysLogHandler) For error message and traceback logging to syslog. Currently supports Windows (via localhost:514), Darwin (/var/run/syslog), Linux (/dev/log) and FreeBSD (/dev/log). You would not be able to access this property if the directory doesn't exist. (Notice that in Docker you have to enable everything by yourself)
- accessTimedRotatingFile (using logging.handlers.TimedRotatingFileHandler) For requests information logging to file with daily rotation support.
- errorTimedRotatingFile (using logging.handlers.TimedRotatingFileHandler) For error message and traceback logging to file with daily rotation support.

And `filters`:

- accessFilter (using sanic.defaultFilter.DefaultFilter) The filter that allows only levels in `DEBUG`, `INFO`, and `NONE(0)`
- errorFilter (using sanic.defaultFilter.DefaultFilter) The filter taht allows only levels in `WARNING`, `ERROR`, and `CRITICAL`

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

- sanic: Used to log internal messages.
- network: Used to log requests from network, and any information from those requests.

Log format:

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

- 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
```

2.16 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 `127.0.01:42101` 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 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.

2.17.1 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.

2.17.2 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)
```

2.17.3 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 --bind 0.0.0.0:1337 --worker-class sanic.worker.GunicornWorker
```

2.17.4 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()
```

2.18 Extensions

A list of Sanic extensions created by the community.

- 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.
- 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.

2.19 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.

2.19.1 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
```

2.19.2 Running tests

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

```
tox
```

See it's that simple!

2.19.3 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.

2.19.4 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.

2.19.5 Warning

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

CHAPTER 3

Module Documentation

- genindex
- search