Sanic Documentation

Release 0.3.0

Sanic contributors

Sep 06, 2018

Contents

|Join the chat at https://gitter.im/sanic-python/Lobby| |Build Status| |PyPI| |PyPI version|

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 aims 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 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:

```
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 used for all requests to that URL. However, the `@app.route` decorator accepts an optional parameter, `methods`, which restricts the handler function to the HTTP methods in the given list.

```
from sanic.response import text
```

(continues on next page)

(continued from previous page)

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

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

2.3 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")
  ```

(continues on next page)

(continued from previous page)

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

- `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.

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']
```

(continues on next page)

(continued from previous page)

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

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

2.4 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)
```

2.5 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.

2.5.1 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)
```

2.5.2 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))
```

2.5.3 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.

2.6 Middleware

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.

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")
```

2.6.1 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.

2.6.2 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 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.

2.7.1 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'})
```

2.7.2 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=[])]
```

2.7.3 Using blueprints

Blueprints have much the same functionality as an application instance.

Middleware

Using blueprints allows you to also register middleware globally.

```
@bp.middleware
async def halt_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')
```

2.7.4 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()
```

2.7.5 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')
blueprint_v2 = Blueprint('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)
app.blueprint(blueprint_v2)

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

2.8 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.

2.8.1 Reading cookies

A user's cookies can be accessed `Request` object's `cookie` 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))
```

2.8.2 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
```

2.8.3 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.

2.9 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.

2.9.1 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(), '/')
```

2.9.2 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>')
```

2.9.3 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')
```

2.9.4 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, '/')
```

2.10 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.

2.10.1 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'
```

(continues on next page)

(continued from previous page)

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

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

2.11 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)
```

2.12 Testing

Sanic endpoints can be tested locally using the `sanic.utils` module, which depends on the additional aiohttp library. The `sanic_endpoint_test` function runs a local server, issues a configurable request to an endpoint, and returns the result. It takes the following arguments:

- `app` An instance of a Sanic app.
- `method` *(default `'get'`)* A string representing the HTTP method to use.
- `uri` *(default `'/'`)* A string representing the endpoint 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.
- `loop` *(default `None`)* The event loop to use.
- `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 with a GET request, `method` would be `get` and the keyword argument `params={'value', 'key'}` would be supplied. More information about the available arguments to aiohttp can be found in the documentation for ClientSession.

Below is a complete example of an endpoint test, using pytest. The test checks that the `/challenge` endpoint responds to a GET request with a supplied challenge string.

```
import pytest
import aiohttp
from sanic.utils import sanic_endpoint_test

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

def test_endpoint_challenge():
    # Create the challenge data
```

(continues on next page)

(continued from previous page)

```
    request_data = {'challenge': 'dummy_challenge'}

    # Send the request to the endpoint, using the default `get` method
    request, response = sanic_endpoint_test(app,
                                            uri='/challenge',
                                            params=request_data)

    # Assert that the server responds with the challenge string
    assert response.text == request_data['challenge']
```

2.13 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).
- `before_start` *(default `None`)*: Function or list of functions to be executed before the server starts accepting connections.
- `after_start` *(default `None`)*: Function or list of functions to be executed after the server starts accepting connections.
- `before_stop` *(default `None`)*: Function or list of functions to be executed when a stop signal is received before it is respected.
- `after_stop` *(default `None`)*: Function or list of functions to be executed when all requests are complete.
- `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.

2.13.1 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.

2.13.2 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)
```

2.14 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.
- Jinja2: Support for Jinja2 template.

2.15 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.

2.15.1 Running tests

- `python -m pip install pytest`
- `python -m pytest tests`

2.15.2 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.

2.15.3 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